Launching MPW Faster • Customizing QuickDraw GX Drivers • Choosing a CODEC

develop

The Apple Technical Journal

Issue 21 March 1995

Getting Started With OpenDoc Graphics

A First Look at Dylan: Classes, Functions, and Modules

Designing a Scripting Implementation

An Object-Oriented Approach to Hierarchical Lists

Introducing PowerPC Assembly Language

$10.00

develop

EDITORIAL STAFF

ART & PRODUCTION

ISSN #1047-0735.


Printed on recycled paper

THINGS TO KNOW

develop, The Apple Technical Journal, a quarterly publication of Apple Computer's Developer Press group, is published in March, June, September, and December. *develop* articles and code have been reviewed for robustness by Apple engineers.

This issue's CD. Subscription issues of *develop* are accompanied by the *develop Bookmark* CD. The Bookmark CD contains a subset of the materials on the monthly *Developer CD Series*, which is available from APDA. Included on the CD are this issue and all back issues of *develop* along with the code that the articles describe. The *develop* code is updated when necessary, so always use the most recent CD. The CD also contains Technical Notes, sample code, and other useful documentation and tools (these contents are subject to change). Software and documentation referred to as being on this issue's CD are located on either the Bookmark CD or the Reference Library or Tool Chest edition of the *Developer CD Series*. The *develop* issues and code are also available on AppleLink and via anonymous ftp at ftp.info.apple.com, in the Developer Services area.

Macintosh Technical Notes. Where references to Macintosh Technical Notes in *develop* are followed by something like "(QT 4)," this indicates the category and number of the Note on this issue's CD. (QT is the QuickTime category.)

E-mail addresses. Most e-mail addresses mentioned in *develop* are AppleLink addresses; to convert one of these to an Internet address, append "@applelink.apple.com" to it. For example, DEVELOP on AppleLink becomes develop@applelink.apple.com on the Internet. To convert a NewtonMail address to an Internet address, append "@online.apple.com" to it.

CONTACTING US

Feedback. Send editorial suggestions or comments to Caroline Rose at AppleLink CROSE, Internet crose@applelink.apple.com, or fax (408)974-6395. Send technical questions about *develop* to Dave Johnson at AppleLink JOHNSON.DK, Internet dkj@apple.com, CompuServe 75300,715, or fax (408)974-6395. Or write to Caroline or Dave at Apple Computer, Inc., One Infinite Loop, M/S 303-4DP, Cupertino, CA 95014.

Article submissions. Ask for our Author's Guidelines and a submission form at AppleLink DEVELOP, Internet develop@applelink.apple.com, or fax (408)974-6395. Or write to Caroline Rose at the above address.

Subscriptions and back issues. You can subscribe to *develop* through APDA (see below) or use the subscription card in this issue. You can also order printed back issues. For subscription changes or queries or back issue orders, call 1-800-877-5548 in the U.S., (815)734-1116 outside the U.S. Or write AppleLink DEV.SUBS or Internet dev.subs@applelink.apple.com. *Be sure to include your name, address, and account number as it appears on your mailing label in all correspondence related to your subscription.* One-year U.S. subscription price is $30 for 4 issues of *develop* and the *develop Bookmark* CD; all other countries, $50 U.S. For Canadian orders, price includes GST (R100236199). Back issues are $13 each in the U.S., $20 all other countries.

APDA. To order products from APDA or receive a catalog, call 1-800-282-2732 in the U.S., 1-800-637-0029 in Canada, (716)871-6555 internationally, or (716)871-6511 for fax. Order electronically at AppleLink APDA, Internet apda@applelink.apple.com, CompuServe 76666,2405, or America Online APDAorder. Or write APDA, Apple Computer, Inc., P.O. Box 319, Buffalo, NY 14207-0319.

ARTICLES

COLUMNS

EDITOR'S NOTE

CAROLINE ROSE

From time to time people I know outside of Apple ask me what kind of Macintosh they should buy for home use. I in turn always ask what made them decide on a Macintosh in the first place. The answer is usually along the lines of "My kid has one at school and loves it" or "I use PCs at work but write my memos on a Macintosh, and I love my Mac." Typically they can't pinpoint the reasons for this "love." People enjoy using the Macintosh; you might say they're charmed by it.

Charm sells. I used to think my taste for older houses with all their nooks and crannies — and yes, imperfections — would work to my benefit in the real estate market. But in fact it seems that's what everyone wants. The houses that suit me are rarely put up for sale, their owners are so loathe to part with them; on those few occasions that they are on the market, they're sold in the blink of an eye. Newer, bigger houses that go for the same price sell much more slowly.

So when I hear about how some new computer is expected to run infinitesimally faster than some other one, I'm not swayed. (You'd be amazed to learn the creaky model of the Macintosh I use at home for my personal tasks.) Through years of complaints about how slow and otherwise imperfect the Macintosh was, I just *knew* it would thrive. I don't think people in the home market, especially, are going to focus on performance measurements or the number of applications available. Of course they need reasonable speed and the necessary applications to do what they want to do — but most of all they want a computer they'll enjoy using. They ask around, and they see where people's hearts lie in the computer-using world. Not that there won't be heartless millions choosing those other computers, but there will *always* be Macintosh.

To quote from Tim Maroney's first installment of "MPW Tips and Tricks," in this issue: "I don't use my computer to run Dhrystone benchmarks: I use it to accomplish tasks."

As someone on the original Macintosh team, I'm not surprised by its appeal to the heart. We all succumbed very early to its charm. I remember the big meeting we had to decide the computer's name, "Macintosh" being a code name that we were resolved not to keep. But no other ideas for names — what few there were — gained headway. The reason "Macintosh" stuck wasn't because it was the name of an apple (if misspelled); it stuck because we'd all grown so fond of our little "Mac." It would have been like renaming our first born.

Call me sentimental; I can take it. Call this my valentine to the Macintosh.

Caroline Rose
Editor

CAROLINE ROSE (AppleLink CROSE) started working at Apple in 1982 the first time, then again in 1991. In between, she learned what it's like to be among the first employees in a startup company run by Steve Jobs. She worked as a programmer back when any math major could pick it up pretty easily and when there were about three programming languages to choose from. Now that there are OODLs of languages out there, she's happy to be back to writing in English. Caroline stood out as an odd bird in grade school because she actually enjoyed diagramming sentences. (There are other reasons, too, but we won't go into those.)•

LETTERS

BOOKMARK CD ALIAS PROBLEM

On the *develop* Issue 19 Bookmark CD, there's an alias in the OpenDoc folder that can't be opened because it apparently thinks it's supposed to be on a disc with a different name than "Bookmark CD 19." What's the problem?

— Eric Shepherd

There was an alias (to "AppleScript™ 1.1") in the OpenDoc A6 folder that pointed to the Developer CD. This slipped by us on the Issue 19 Bookmark CD; it should have pointed to a file explaining that the software was on the Developer CD or could be obtained through APDA. (Occasionally, we're unable to publish certain software packages on the Bookmark CD.) Thanks for pointing this out; we're now checking more carefully for such things.

— Alex Dosher

HOW NOT TO DO PREFERENCES?

When I saw the article on writing preferences files in Issue 18, I thought it would be great to finally have someone explain how to do it properly. But the article didn't really do that, at least not in my opinion, and it didn't agree with what Apple software does.

For one thing, I really think preferences files should use the 'pref' file type; it keeps down the time it takes for the Finder to display the contents of the Preferences folder, it automatically gives the files the correct icon, and it avoids the foolishness of having to register two creators for every application. The Finder should be revised so that double-clicking a file of type 'pref' gives the "can't open preferences file" alert.

Programmers should not need to put what is effectively a fixed string in every preferences file. Balloon Help on a file of type 'pref' should yield something sensible instead of erroneously naming the file as the Finder's preferences file, especially since many other preferences files already use the 'pref' type.

Most of the rest of the article was on target, especially the bit about not putting static data in the preferences folder. But there was no comment about where static, shared data should go. The trend seems to be to drop everything into the Extensions folder, but it would be better if a new "Data" or "Shared" folder were created in the System Folder to support these shared resources.

— Peter N. Lewis

I find little to disagree with in your note. I've received other messages on this subject, many containing the same good suggestions, among others.

I'll be the first to admit that several of the methods I describe in the article are, in their best light, convoluted, and at worst, an unpleasant hack. The article reviewers and I discussed the issues with Apple engineers and human interface folks, and nobody could come up with a better solution for the system as it exists today. (Maybe the title should have been "A Good Way to Implement Preferences Files" instead of "The Right Way . . .") I tried to codify the current thinking at Apple on the best way to handle preferences files in the existing system software environment, and I think I achieved that goal.

I'm actually pleased (in a perverse sort of way) that the article has created somewhat of a stir: it highlights the problems in this

IF YOU WRITE, WE WILL ANSWER

We welcome timely letters to the editors, especially regarding articles published in *develop*. Letters should be addressed to Caroline Rose — or, if technical *develop*-related questions, to Dave Johnson — at AppleLink CROSE or JOHNSON.DK. Or you can write to Caroline or Dave at Apple Computer, Inc., One Infinite Loop, M/S 303-4DP, Cupertino, CA 95014. All letters should include your name and company name as well as your address and phone number. Letters may be excerpted or edited for clarity (or to make them say what we wish they did).•

area faced by developers better than anything I might write. Granted, in the overall scheme of things preferences files are rather low on the totem pole, but still, this discussion may provide some impetus at Apple to fix things in a subsequent system release. At that time, I'll be more than happy to revisit the topic and do away with the two-creator hack forever!

It was also pointed out to me that the preferences library doesn't include any provision for handling cross-platform issues, and that because it's resource-based, it precludes the possibility of being cross-platform. What can I say? This is what happens when your thinking is too Mac-centric. Even though I don't work at Apple any more, I'm still a hopeless Mac fanatic!

— Gary Woodcock

OBJECT-ORIENTED LISTS

The article on hierarchical lists in Issue 18 was especially interesting to me because I'm writing an object-oriented database and need to display large amounts of hierarchically organized objects. I'm using the PowerPlant library LListBox class; however, I want to be able to display icons and triangular buttons along with styled text, so I've started adding LDEF modifications to achieve that. The LDEF approach, being ultimately based on the List Manager, will have problems with large lists. Any suggestions you may have would be greatly appreciated.

— Maynard Chen

You're in luck; we just happen to have a followup article with the information you need. See "An Object-Oriented Approach to Hierarchical Lists" in this issue.

— Caroline Rose

FLOATING WINDOWS UPDATE

Recently, when I tried to use the floating windows library that was described in *develop* Issue 15, I ran into a few problems getting it to work with the universal headers (the library relies on SysEqu.h for the existence of the WindowList global variable). I'm trying to recompile this library for use with my CodeWarrior projects (MPW and I have had a falling out over speed!) but I'm having problems. Is there a more recent version of the code than the one on the June 1994 CD? If not, can you tell me how to fix it?

— David A. denBoer

An updated version of the floating windows code appears on this issue's CD. The only changes to the code were to replace SysEqu.h with LowMem.h in the includes, and to change GetWindowList and SetWindowList to use the new low-memory accessor routines. The sample application was also revamped slightly to compile in CodeWarrior.

— Dave Johnson

SPOTTED DICK REVEALED

The British dessert "spotted dick" seems to have become a running joke in *develop*. You've got me curious. Can your technical staff do some research and give us the scoop on this?

— Steven C. Johnson

Spotted dick is Reason #87 on the list of Why There Will Always Be an England. (Reason #112 is "The word Worcester is pronounced Wooster.") A dick is a steamed dessert cake, or "pudding," made of suet (or shortening), flour, and other ingredients (like sugar) to make it taste good. It's usually served hot, with a milky syrup the British refer to as custard. If you add currants to the recipe, the dick ends up having spots, hence the name "spotted dick."

In a recent edition of The Patrick O'Brian Newsletter (he being a favorite author of mine), I was surprised to see a reference to this dessert as Spotted Dog, along with the variations Drowned Baby (glutinous surface), plum duff (prunes), figgy-dowdy (raisins), and roly-poly (rolled and spread with jam).

Surely this is more than you ever wanted to know. But for the terminally curious, I've got an actual recipe. Douglas Norton (of — you guessed it — Great Britain) sent it in "just to show that someone does read the little pieces at the bottom of the page."

— Caroline Rose

Getting Started With OpenDoc Graphics

The layout and imaging services offered by OpenDoc, Apple's compound-document architecture, provide extremely powerful support for document layout. However, with power comes a certain amount of complexity. The introduction to OpenDoc graphics given in this article reduces some common graphics operations to simple recipes. By following these recipes, you'll get a sense of how to use OpenDoc that you can later build on as you learn about its more sophisticated capabilities.

KURT PIERSOL

OpenDoc's layout and graphics model is designed to allow maximum flexibility at imaging time. You can use it to create very complex displays that include real-time motion, offscreen rendering and compositing, and more. But at first glance, it can appear complicated and bewildering. Don't despair: the good news, as you'll learn in this article, is that you can use it for simple tasks without much trouble.

I touched on some of the basics of the OpenDoc layout model and drawing code in my article "Building an OpenDoc Part Handler" in *develop* Issue 19 (which you need not have read before reading this article). Here I reiterate a little of that and add to it as I explain the basic terminology and concepts of OpenDoc graphics. Then I present a series of simple recipes (also provided on this issue's CD) that illustrate the use of OpenDoc graphics objects. You'll learn how to draw a part, scroll the part, zoom or rotate the part's content, make an embedded part visible, alter the coordinate system scaling, and do simple printing under QuickDraw.

THE BASICS OF OPENDOC GRAPHICS

OpenDoc objects work together to lay out and draw each piece of content (each *part*) in a compound document. We'll take a look at the layout model here before focusing on each of its constituent objects and how the objects relate to one another.

THE LAYOUT MODEL

OpenDoc's layout model includes both a persistent representation and a runtime representation of a document's state. Persistent information is represented in objects called *frames*, while runtime information is captured in objects called *facets*. The two sets of objects, working together, produce the structure of the displayed or printed document.

KURT PIERSOL is a system architect at Apple and has been involved with the Apple events project, AppleScript, and OpenDoc. You can recognize him by his eccentric fashion sense and his tendency to use funny accents during heated engineering discussions.•

Early releases of OpenDoc will be made available through a number of different sources, including *develop*.•

Frames are arranged in a lattice (speaking in mathematical, not geometric, terms). Any frame can contain any other, but in practice they almost always fall into a strict hierarchy, with each frame contained in only one other frame. Frames always contain a pointer to their containing frame but not directly to their embedded frames. Some applications — like Personal Information Managers, which handle lots of unstructured information — have more sophisticated data models, however, so OpenDoc is built to accommodate these applications.

Facets are *always* arranged in a strict hierarchy, and every facet has pointers to every contained facet as well as to the containing facet. OpenDoc walks this structure at run time to perform drawing as well as to handle geometric events such as mouse clicks.

The runtime representation is hooked into the window system by means of a window object. This window object simply points to the topmost facet in that window's facet hierarchy.

To understand these objects in greater detail, you need to be familiar with three basic ideas that form the foundation of OpenDoc's layout and imaging capabilities: canvas, shape, and transform.

- A *canvas* is simply a drawing context. Different platforms have different ideas of what a canvas might be, based on the particular graphics toolbox they provide. On the Macintosh, a canvas can be either a QuickDraw graphics port or a QuickDraw GX view port.
- A *shape* is a way to describe an area of a canvas. OpenDoc provides a platform-independent shape definition based on polygons. In addition, for speed, QuickDraw regions and rectangles and QuickDraw GX shapes can serve as shape objects in OpenDoc.
- A *transform* is a way of altering the coordinate system that applies to a particular canvas. You're familiar with the concept of transforms if you provide scrolling in your applications. When a window is scrolled, a new origin is set in the graphics port before drawing calls are performed. This offset is an example of a transform. In QuickDraw, the only transforms with built-in support are offsets and (partly) scaling, both of which require a certain amount of work on the programmer's part; QuickDraw GX offers offsets and scaling as well as more interesting transforms such as rotating and skewing. OpenDoc supports full two-dimensional transformations and provides hooks to supply your own transform types.

With these definitions in mind, let's take a closer look at frames, facets, and windows.

FRAMES

A containing part and its embedded part share a single frame object, which they use to communicate persistent layout information. Much of this information is communicated in the form of shapes, which are ways to describe a geometric area. Specifically, layout information is communicated by way of the frame shape, the used shape, and the internal transform associated with a frame.

The *frame shape* is how the container tells the embedded part how much area it has in which to lay itself out. The container "owns" this shape, meaning that it's allowed to set the value. If the embedded part wants a different frame shape, it must ask the container to change it, and the container might refuse.

The rule in this sort of negotiation is "Don't ask twice!" This means that if an embedded part asks for a new frame shape and is denied, it shouldn't ask again. It might want to request a different shape later, but it should never again request exactly the same shape. The reason, as you might imagine, is to avoid infinite loops in negotiation. Very long, dull, and useless negotiations between parts may well result if the "Don't ask twice!" rule isn't followed.

The *used shape* serves to inform the container exactly what part of the frame shape the embedded part decided to use, and is "owned" by the embedded part. For example, imagine that a word processing container passes a rectangular area as the frame shape, but an embedded pie chart uses only a circular area within the frame shape. The pie chart can inform the word processor of this by setting the used shape of the frame to the circular area actually used. It's then the responsibility of the container to fill in any unused areas of the frame with an appropriate background or drawing. This is what makes part transparency work.

The *internal transform*, also "owned" by an embedded part, captures information about how the embedded part wishes to transform its content when it's displayed. If, for example, the pie chart of the previous example were too big to fit in the allotted space in the word processor and wanted to scroll itself to a particular location, it could do so by setting the internal transform of the frame. We'll look at some examples in the recipes section to come.

FACETS

A facet is similar to a QuickDraw GX view port, or to a QuickDraw graphics port on steroids; it's a description of the place where a particular frame of a part becomes visible. Typically passed in to your part handler by an object at drawing time, a facet has information about where the part should be drawing the content of the frame right now. It specifies the canvas where all drawing calls should be made and also includes clipping and transformation information.

The clipping information appears in the form of the *clip shape*. This shape specifies exactly where on a canvas a part handler can draw. It's equivalent to the content region of a window in a traditional Macintosh application. Actually, there are two versions of this shape that you can retrieve: the aggregate clip shape, which is the clipping information relative to your drawing canvas, and the clip shape, which is relative to the coordinate system of your container. Your container owns the clip shape and sets it in its own coordinate system. You should always clip any drawing calls you make to the aggregate clip shape of the facet you're drawing into.

The transformation information comes in the form of a set of transform objects that are available from the facet, each one specifying a particular coordinate system related to the frame being displayed. The *external transform* of a facet specifies where the facet sits in relation to its container. For example, as illustrated in Figure 1, if the embedded facet should be offset by (100,100) from the origin of its container, the external transform would specify an offset of (100,100). Note that in this figure, only a portion of the content is being displayed in the embedded facet, and the coordinates (0,0) refer to the content and indicate that the content's upper left corner presently coincides with the upper left corner of the facet.

The internal transform of a frame is composed with the external transform of its facet plus all the transforms above it in the facet hierarchy to give the complete transformation used to draw into the facet, called the *content transform*. In Figure 2, which illustrates this process of composition, both transforms are simply offsets. The external transform specifies an offset of (100,100), so the embedded facet is offset by (100,100) from the origin of its container, the same as in Figure 1. The internal

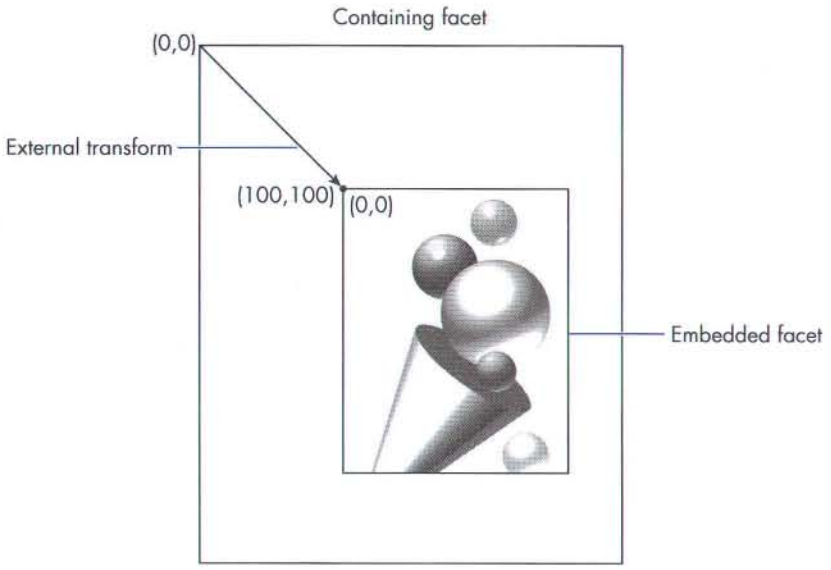


Figure 1. An embedded facet with an external transform of (100,100)

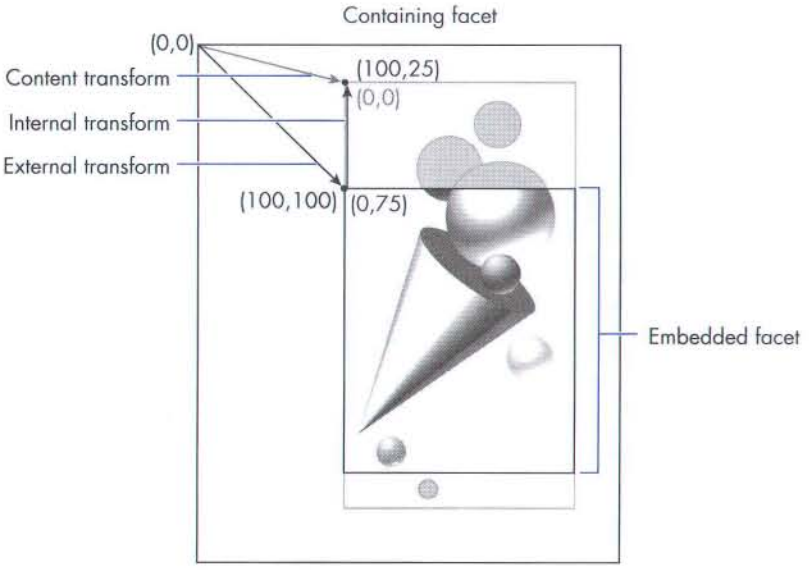


Figure 2. Composing internal and external transforms to obtain a content transform

transform specifies an offset of (0,-75), so the content is scrolled upward from its position in Figure 1. A different portion of it now shows in the embedded facet, which remains the same size as before.

This composition is recursive, so every level of embedding adds a new external and internal transform to the final transformation. All of the transforms must be composed together to produce a correct graphical result. When drawing occurs, the content transform should be applied to any drawing commands. The only exception to this rule is if you're drawing content that shouldn't scroll as well as content that should. In this case, use the content transform for scrolling content and use a different transform, called the *frame transform*, to draw the rest. The frame transform is exactly the same as the content transform except that it doesn't include the internal transform of the innermost frame.

WINDOWS

Windows are the objects that hook OpenDoc facets and frames into the Macintosh window structure. A window object holds a Macintosh window structure, as well as a pointer to the topmost facet visible in the window. We call this facet the *root facet* because it's the root of the facet hierarchy in that window. The frame being displayed through that facet is called the *root frame*, and the part being displayed in the root frame is called the *root part*.

HOW THESE OBJECTS RELATE TO ONE ANOTHER

When a window is visible, it points to a facet at the root of the window. The graphics port, or root QuickDraw GX view port, of that window is used as the canvas on which that facet appears.

Every facet displays a particular frame, but a frame can be visible in more than one facet at the same time. Every frame displays a particular part, but a part can be displayed in more than one frame, as shown in Figure 3. Here we see a window displaying two parts: a drawing container that has chosen to split itself into two independently scrollable sections, and an embedded charting part. To split itself, the container has set up two facets on the same frame of its embedded charting part, as indicated in the schematic to the right of the window (in which the arrows represent pointers). This automatically causes the embedded charting part to display synchronized views of itself in the two scrollable sections. This is the model that people who already do splitting in their code are most likely to implement, although there are more elegant models included in the standard recipes that are part of the OpenDoc Software Development Kit.

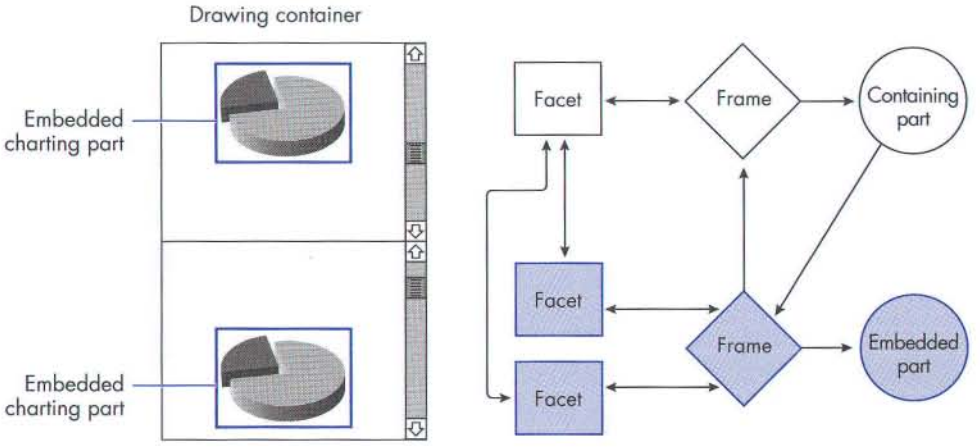


Figure 3. A split window using multiple facets of the same frame

When the information in a part changes, the part needs to update every frame in which it's displayed. When a part updates the information in a frame, it should redraw into every facet of the frame. Let's assume, for example, that we have a charting part that can display a bar chart. When someone changes a data value, the part should make sure that every frame is redrawn. Since every frame may have multiple facets, the part handler should iterate through every display frame, iterate through each frame's visible facets, and draw the content into each. To accomplish this, you can decide to invalidate all of the affected frames and let OpenDoc make sure everything is redrawn correctly. However, if performance or timing constraints make this impractical, or if flicker is an issue, your part handler can draw directly into each facet or each affected frame, using a doubly nested loop.

SOME OPENDOC GRAPHICS RECIPES

Now let's look at a series of examples of how to use these objects to perform basic graphics operations. Each of the following recipes is, by nature, just a skeleton. Every part handler has different drawing code, so we'll concentrate on outlining general recipes and wave our hands over the specific drawing commands.

The examples are all based on Color QuickDraw, on the assumption that more readers will be familiar with these calls than with the QuickDraw GX equivalents. I'm counting on you QuickDraw GX aficionados to perform the necessary mapping between QuickDraw and QuickDraw GX as you read. The QuickDraw GX equivalent calls will work equally well to clip, set up drawing contexts, and perform geometric operations. Of course, you'll find QuickDraw GX helps tremendously in implementing scaling, rotation, and other transformations. Even though these examples are in Color QuickDraw, I urge you to use QuickDraw GX as your basic imaging model if you can.

The example code you'll see here is a simplified version of code written by Steve Smith and Eric House. The good ideas are theirs; the mistakes are probably mine. This is not intended to be working code, since error handling and some other pieces have been left out for the sake of simplicity.

DRAWING A PART

Our first recipe tells how to draw a part. There are only a few simple steps here, for the most basic case. Take a look at Listing 1 as we discuss the recipe. To keep things as simple as possible, we'll ignore scrolling and printing for the moment. The basic steps are as follows:

1. Get the canvas of the facet you're drawing into and set the graphics port to that.
2. Get the content transform out of the facet and use it to set offset information.
3. Get the aggregate clip shape out of the facet and use it to set the clipping region.
4. Draw your content.
5. Clean up.

As you may have noticed, this is only slightly more complex than using the existing window system. Luckily, all of the added complexity comes only in the setup code, not in the actual drawing calls. Once the drawing environment has been set up, you're free to make the same drawing calls you always have.

Incidentally, there's an easy way to get all of the correct setup code done for you: a public utility called FocusLib, available as part of the OpenDoc Software Development Kit, reduces the setup code you see in this and the following examples to a single call. I didn't use it here because I wanted you to see exactly what needs to be done, just in case FocusLib doesn't meet your precise requirements and you must do it yourself. For instance, FocusLib is in C++, so not everyone can use it.

SCROLLING THE PART

Creating a scrolling part is only a bit more complex than making one that doesn't scroll. The best way to scroll in OpenDoc is to modify the internal transform of your part's display frame. This has the effect of automatically scrolling any embedded parts that are visible right now, without any extra work on your part. Check out Listing 2,

Listing 1. MyPart::Draw, simplest case

```
void MyPart::Draw(Environment* ev, ODFacet* facet, ODShape* invalShape)
{
   // Set up graphics port.
   GrafPtr port = facet->GetCanvas(ev)->GetQDPort(ev);
   SetPort(port);

   // Set up graphics port offset for drawing content.
   ODTransform* localToGlobal = facet->GetContentTransform(ev, kODNULL);
   ODPoint offset(0,0);
   offset = localToGlobal->TransformPoint(ev, &offset);
   SetOrigin(-offset.IntX(), -offset.IntY());
   localToGlobal->Release(ev);

   // Set up graphics port clip; save old clip.
   RgnHandle saveClip = NewRgn();
   GetClip(saveClip);
   ODShape* clipShape = facet->GetAggregateClipShape(ev, kODNULL);
   RgnHandle clip = clipShape->GetQDRegion(ev);
   SetClip(clip);
   clipShape->Release(ev);

   // And draw (insert your drawing code here).
   // Remember to respect the scaling and rotation information
   // in your content transform, if possible. If you can't do this,
   // it's graceful to at least try to draw as best you can, rather
   // than simply signaling an error.
   ...

   // Clean up.
   SetClip(saveClip);
   DisposeRgn(saveClip);
   SetOrigin(0,0);
}
```

which shows the changes in the draw method for this slightly more complex case. The basic steps are as follows:

1. Get the canvas of the facet you're drawing into and set the graphics port to that.
2. Get the frame transform out of the facet and use it to set offset information.
3. Get the aggregate clip shape out of the facet and use it to set the clipping region.
4. Draw your scroll bar control.
5. Get the content transform out of the facet and use it to set offset information.
6. Make sure the scroll bar area is removed from the clipping region.
7. Draw your content.
8. Clean up.

Listing 2. MyPart::Draw, with scrolling

```
void MyPart::Draw(Environment* ev, ODFacet* facet, ODShape* invalShape)
{
   Point    spclOffset = {0,0};
   Point    contentOffset = {0,0};

   // Set up graphics port.
   GrafPtr port = facet->GetCanvas(ev)->GetQDPort(ev);
   SetPort(port);

   // Set up graphics port offsets for controls.
   ODTransform* localToGlobal = facet->GetFrameTransform(ev, kODNULL);
   ODPoint tempOffset1(0,0);
   tempOffset1 = localToGlobal->TransformPoint(ev, &tempOffset1);
   SetOrigin(-tempOffset1.IntX(), -tempOffset1.IntY());

   // Set up special offset for later efficient reset of clipping region.
   spclOffset = localToGlobal->GetQDOffset(ev);
   ODTransform* contentTransform = facet->GetContentTransform(ev,
         kODNULL);
   contentOffset = contentTransform->GetQDOffset(ev);
   spclOffset.v -= contentOffset.v; spclOffset.h -= contentOffset.h;
   localToGlobal->Release(ev);

   // Set up graphics port clipping.
   RgnHandle saveClip = NewRgn();
   GetClip(saveClip);
   ODShape* clipShape = facet->GetAggregateClipShape(ev, kODNULL);
   RgnHandle clip = clipShape->GetQDRegion(ev);
   SetClip(clip);

   // Draw your controls (such as scroll bars) here, using the standard
   // Toolbox calls. Remember to respect the scaling and rotation
   // information in your transform, or at least try to draw as best
   // you can, rather than simply signaling an error.
   ...

   // Set up graphics port for drawing content.
   ODPoint tempOffset2(0,0);
   tempOffset2 = contentTransform->TransformPoint(ev, &tempOffset2);
   SetOrigin(-tempOffset2.IntX(), -tempOffset2.IntY());

   // Use the special offset we set up earlier to keep the clip in the
   // right place relative to the origin. Remember to remove the scroll
   // bar area from the clipping region.
   OffsetRgn(clip, spclOffset.h, spclOffset.v);
   SetClip(clip);

   // And draw (insert your drawing code here).
   // Remember to respect the scaling and rotation information
   // in your content transform if possible, as mentioned above.
```

(continued on next page)

Listing 2. MyPart::Draw, with scrolling *(continued)*

```
    ...

    // Clean up.
    SetClip(saveClip);
    DisposeRgn(saveClip);
    contentTransform->Release(ev);
    clipShape->Release(ev);
    SetOrigin(0, 0);
}
```

Listing 3 shows how you would handle the actual scrolling. The recipe here is also quite simple. By altering the internal transform of the frame being displayed, you change not only the display of the content but also the position of all embedded facets. For this recipe to work, the previous recipe must have been implemented. The steps are as follows:

1. Track the scroll bar control.
2. Decide the scrolling distance.
3. Create a transform that offsets by the correct amount.
4. Call ScrollRect; invalidate the correct areas of the screen.
5. Change the internal transform of the frame being scrolled.

You can no doubt imagine much more complex scrolling behavior, with significantly better optimization than you see in these examples.

There's a bit of human interface to mention at this juncture. You shouldn't grab the selection focus simply because scrolling is occurring. At first blush, it might make sense to imagine that a part should become active when scrolling. There's no particular reason to do this, though. Scroll bars should become active when the window is frontmost, but a part should become active (that is, grab the selection focus) only when a selection is made within it.

ZOOMING OR ROTATING THE PART'S CONTENT

The process of zooming or rotating your content is closely related to the scrolling recipe we just examined. Again, the internal transform can be used to apply to not only your own content but that of embedded parts as well. Listing 4 is a simple example of a 4x scaling operation that zooms in by a factor of 4. As before, it assumes correct behavior from the drawing code, as illustrated in Listing 1.

The recipe for both zooming and rotating is fairly simple:

1. Set up a transform that captures the desired transformation.
2. Change the internal transform of the frame.
3. Invalidate the frame to make sure it gets redrawn.

MAKING AN EMBEDDED PART VISIBLE

When you're writing a handler for parts that can contain other parts, there comes a time when you have a pointer to a part that hasn't yet been made visible. We won't go into exactly how such a part is obtained from the Clipboard or the drag and drop

Listing 3. Handling the scrolling

```
ODBoolean MyPart::HandleMouseDownScrollBar(Environment* ev, Point mouse,
      ODFrame* frame)
{
   ODSShort partcode = TrackControl(fScrollBar, mouse, kODNULL);
   Point transPt;
   ODSShort setting = GetControlValue(fScrollBar);

   if (partcode) {
      // Deal with the scroll bar and choose the scroll distance.
      switch (partcode) {
         case inUpButton:
            setting--;
            break;
         case inDownButton:
            setting++;
            break;
         case inPageUp:
            if ((setting - kPBPageSize) < 0) setting = 0;
            else setting -= kPBPageSize;
            break;
         case inPageDown:
            ODSShort max = GetControlMaximum(fScrollBar);
            if ((setting + kPBPageSize) > max) setting = max;
            else setting += kPBPageSize;
            break;
         default:
            break;
      }

      SetControlValue(fScrollBar, setting);
      SetPt(&transPt, 0, (0-setting));    // This is a vertical
                                          // scroll bar.

      // Set up the transform.
      ODTransform* newIntTrans = frame->CreateTransform(ev);
      newIntTrans->SetQDOffset(ev, &transPt);
      frame->ChangeInternalTransform(ev, newIntTrans, kODNULL);
      newIntTrans->Release(ev);

      // Here's some simplified code for invalidating any embedded
      // facets that have been moved. This is a "saturation bombing"
      // approach rather than the tuned code of a real application,
      // but it gets the basic idea across. Typically, we do a
      // ScrollRect followed by invalidation of the revealed area.
      // Here we simply invalidate everything.
      frame->Invalidate(ev, kODNULL, kODNULL);
      return kODTrue;
   }
   return kODFalse;
}
```

Listing 4. A 4x scaling operation

```
void MyPart::ZoomContents4X(Environment* ev, ODFrame* frame)
{
   ODPoint     frameScale(4, 4);

   // Apply the zoom transformation.
   ODTransform* intTrans = frame->GetInternalTransform(ev, kODNULL);
   intTrans->ScaleBy(ev, &frameScale);
   intTrans->Release(ev);

   // Invalidate the frame for redrawing.
   frame->Invalidate(ev, kODNULL, kODNULL);
}
```

object. Suffice it to say that if you do things correctly, you'll end up with a pointer to a storage unit that has part data inside it.

To embed this part, you must create a frame and a facet for it. Once you've done that, you probably want to remember the frame and the facet for further reference. Take a look at Listing 5 for an example. The recipe for making a part visible and embedding it inside your part is as follows:

1. Get the part, using the draft, from the storage unit pointer.
2. Create a frame to display the part.
3. Create a facet to make the frame visible.
4. Invalidate the facet.
5. Clean up.

Listing 5. Making an embedded part visible

```
void MyPart::EmbedPartFromSU(Environment* ev, ODStorageUnit* newSU,
      ODFacet* myFacet)
{
   ODPart* newPart = newSU->GetDraft(ev)->GetPart(ev, newPartID);
   ODRect rect(0, 0, kPBDefaultFrameSize, kPBDefaultFrameSize);
   ODShape* newFrameShape = myFacet->CreateShape(ev);
   newFrameShape->SetRectangle(ev, &rect);
   ODFrame* newFrame = newSU->GetDraft(ev)->CreateFrame(ev,
         kODNULL,                  // Use the default frame type.
         myFacet->GetFrame(ev),    // Containing frame is my frame.
         newFrameShape,            // Use the frame shape we set up.
         (ODCanvas*)kODNULL,       // No special canvas.
         newPart,                  // The part in the frame.
         fSession->Tokenize(ev, kODViewAsFrame),  // View as a frame.
         kODNullTypeToken,         // Undefined presentation.
         kODFalse,                 // Not a root frame.
         kODFalse);                // Not overlaid.
```

(continued on next page)

Listing 5. Making an embedded part visible *(continued)*

```
    newSU->Release(ev);
    // Set up a clip shape, same as frame shape.
    ODShape* frameShape = newFrame->GetFrameShape(ev, kODNULL);
    ODShape* newClipShape = frameShape->Copy(ev);
    Point offset = {100,100};
    ODTransform* newExternalXForm = myFacet->CreateTransform(ev);
    newExternalXForm->SetQDOffset(ev, &offset);
    ODFacet* newFacet = myFacet->CreateEmbeddedFacet(ev,
          newFrame,           // Of this frame.
          newClipShape,       // With this clip shape.
          newExternalXForm,   // At this location/scale/rotation.
          kODNULL,            // No special canvas.
          kODNULL,            // No special bias canvas.
          kODNULL,            // No sibling specified.
          kODFrameInFront);   // Put this facet at front.

    // At this point, we probably want to remember the new frame
    // and facet in some data structure.
    this->rememberFrame(newFrame, newFacet);

    // Invalidate facet so that it displays. We need to invalidate our
    // entire frame because the new facet may interact with other content
    // in various ways. This is the simplest correct method, although
    // optimizations may be reasonable in production code.
    myFacet->GetFrame(ev)->Invalidate(ev, kODNULL, kODNULL);

    // Clean up.
    newPart->Release(ev);
    newFrame->Release(ev);
    newFrameShape->Release(ev);
    newExternalXForm->Release(ev);
    newClipShape->Release(ev);
    frameShape->Release(ev);
}
```

ALTERING THE COORDINATE SYSTEM SCALING

A common misconception about OpenDoc is that its base coordinate system is insufficient to handle truly large documents. The coordinate system is in fixed-point 16.16 numbers, where the number 72.0 represents 1 inch. If we take this literally, it means that coordinates in this space can only represent an area of about 455 inches on a side and that we can place elements in that area to a precision of about 1/2,000,000 of an inch.

Clearly, we could trade off a little bit of that precision for a larger working area. This would involve scaling any incoming points or shapes. Actually, by doing this, we can easily lay out an area about 2,000 typical pages on a side at a precision of about 1/4,000 of an inch, including some bits for round-off error in the scaling multiplications. You can ask why we chose 72.0 as the basic scale of our coordinate system, but we won't tell you.

The key here is to work internally in coordinates that suit you, without disturbing the coordinate scaling of your embedded parts. The best way to do this is to apply an

internal scaling transform for your own graphics and then apply external scaling transforms to all of your embedded parts.

For example, to get a larger area that still has adequate placement, you might decide to work in a coordinate system where 1.0 = 1 inch. This would still let you place elements with an accuracy of about 1/32,767 of an inch but would let you work in a layout space that was 32,767 inches on a side. This is about 3,000 pages high, a pretty good size. You'll probably get some rounding error during multiplication, though, so it's best not to count on exact placement within more than about 1/1,000 of an inch at this scaling. Listing 6 is an example of how to do this particular transformative trick.

Listing 6. Altering the coordinate system scaling

```
void MyPart::ShiftCoordinateScaling(Environment* ev, ODFrame* frame)
{
   ODPoint     frameScale(72,72);

   // Set up the transform.
   ODTransform* existingTransform = frame->GetInternalTransform(ev,
      kODNULL);
   ODTransform* newIntTrans = existingTransform->Copy(ev);
   existingTransform->Release(ev);
   newIntTrans->ScaleDownBy(ev, &frameScale);

   // Apply the zoom transformation within myself.
   frame->ChangeInternalTransform(ev, newIntTrans, kODNULL);
   newIntTrans->Release(ev);

   // Now, be a good citizen and iterate over every facet of myself.
   ODFrameFacetIterator* facets = frame->CreateFacetIterator(ev);
   for (ODFacet* facet = facets->First(ev); facets->IsNotComplete(ev);
         facet = facets->Next(ev)) {
      // For each facet of myself, find every embedded facet.
      ODFacetIterator* embeddedFacets = facet->CreateFacetIterator(ev,
         kODTopDown, kODFrontToBack);
      for (ODFacet* embeddedFacet = embeddedFacets->First(ev);
            embeddedFacets->IsNotComplete(ev);
            embeddedFacet = embeddedFacets->Next(ev)) {
         // Alter the transform of each embedded facet.
         existingTransform = embeddedFacet->GetExternalTransform(ev,
            kODNULL);
         ODTransform* newExtTrans = existingTransform->Copy(ev);
         // Notice that we scale in the opposite direction in
         // each embedded frame.
         newExtTrans->ScaleBy(ev, &frameScale);
         embeddedFacet->ChangeGeometry(ev, kODNULL, newExtTrans,
            kODNULL);
         newExtTrans->Release(ev);
         existingTransform->Release(ev);
      }
      ODDeleteObject(embeddedFacets);
   }
   ODDeleteObject(facets);
}
```

SIMPLE PRINTING UNDER QUICKDRAW

The last and most complicated recipe is getting a part to print under QuickDraw. As you know, any printing activity on the Macintosh is fairly complex. OpenDoc doesn't add much complexity to the process, as you'll see from the example.

First, every part needs to understand how to draw correctly to a PostScript™ printer. There's a trick to this because of how the LaserWriter drivers operate. They don't clip to regions if asked to do so; instead, they merely clip to the bounding rectangle of the suggested region. So to set a clipping region when printing to a PostScript device, you'll need to create a clipping polygon in PostScript. This is true even for parts that are always rectangular, because they might be clipped irregularly.

You can detect the presence of a PostScript device by examining the printing job at draw time. The printing job is always available in the canvas when you're being asked to draw to a PostScript device. There are a number of ways to check this, but no standard method that Apple supports; typically, you check the device number and make a choice based on a table kept in your code. If you are in fact drawing to a PostScript device, call the routines shown in Listing 7, rather than SetClip, to set clipping for your part. You don't have to do this when you're printing to a non-PostScript printer, because simple region clipping works just fine there.

You may well want to consider using ColorSync during printing. There's nothing special to worry about in OpenDoc; just go ahead and take advantage of it as desired. Other parts may be using ColorSync independently when they draw as well, and all of these may use independent color-matching methods.

Every part should understand how to perform printing of the window if it's the root part of that window. You'll invoke this recipe when the Print command is chosen from the Document menu. The recipe presented here works for rather simple parts but probably not for complex parts like full word processors or page layout programs that have hundreds of pages to lay out and print. In addition, to simplify the look of the code, a special helper object handles most of the direct calls to the printer driver. Finally, this part doesn't alter its layout for printing, as some parts might want to do.

Given these caveats, though, the basic recipe is as follows:

1. Create a printing job.
2. Use the graphics port from the printing job to create a canvas.
3. Add a facet to your root frame on the canvas.
4. Loop through copies and pages.
    a. Reset the offset and clipping of the facet so that the correct page is showing on the canvas.
    b. Invalidate the facet to cause it to draw.
5. Clean up.

Listings 8 through 10 contain some code to help you implement this recipe. Rather than attempt to show you an entire printing loop here, I'll show three routines that correspond to the key differences in the OpenDoc model of printing.

Listing 8 shows how to perform steps 2 and 3 of the recipe above. This routine expects to be passed a printing port, the page rectangle for the first page of the print job, and the root frame of the window being printed. The routine takes this information and creates a facet on the root frame. It then builds a canvas on the print job's printing port and sets the facet to draw on that canvas. We do this so that we can

Listing 7. Setting clipping when printing to a PostScript device

```
#define kPostScriptBegin   190   // Picture comments for PostScript
#define kPostScriptEnd     191   // printing.
#define kPostScriptHandle  192

void ODBeginPostScriptClip(Environment* ev, ODShape *shape)
{
   ODPolygon poly = shape->CopyPolygon(ev);
   Handle clipHandle = NewEmptyHandle();

   AppendString("\pgrestore", clipHandle);  // Utility routine to append
   AppendString("\pnewpath", clipHandle);   // string to a handle.

   char buf[128];
   ODContour *cont = poly.FirstContour();

   for (long n=poly.GetNContours(); n>0; n--) {
      const ODPoint *v = cont->vertex;
      long m = cont->nVertices;
      if (m > 2) {
         sprintf(buf, "%.2f %.2f moveto", v->x/65536.0, v->y/65536.0);
         AppendBuf(buf, strlen(buf), clipHandle); // Utility routine to
                                    // append string buffer to a handle.
         while (--m > 0) {
            v++;
            sprintf( buf, "%.2f %.2f lineto", v->x/65536.0,
               v->y/65536.0);
            AppendBuf(buf, strlen(buf), clipHandle);
         }
      }
      AppendString("\pclosepath clip", clipHandle);
      cont = cont->NextContour();
   }

   // Set pic comment; then delete clipHandle.
   AppendString("\pgsave", clipHandle);
   PicComment(kPostScriptBegin, 0, kODNULL);
   PicComment(kPostScriptHandle, GetHandleSize(clipHandle), clipHandle);
   PicComment(kPostScriptEnd, 0, kODNULL);
   DisposeHandle(clipHandle);
}

void ODEndPostScriptClip( )
{
   Handle clipHandle = NewEmptyHandle();
   AppendString("\pgrestore", clipHandle);
   PicComment(kPostScriptBegin, 0, kODNULL);
   PicComment(kPostScriptHandle, GetHandleSize(clipHandle), clipHandle);
   PicComment(kPostScriptEnd, 0, kODNULL);
   DisposeHandle(clipHandle);
}
```

Listing 8. Setup for basic printing

```
ODFacet* MyPart::BeginPrinting(Environment *ev, ODFrame* rootFrame,
      TPrPort* thePrPort, ODRect *pageRect)
{
   // Set up identity transform, get page rect, set up to clip to it.
   ODTransform* xtransform = rootFrame->CreateTransform(ev);
   ODShape* clipshape = rootFrame->CreateShape(ev);
   clipshape->SetRectangle(ev, pageRect);

   // Create a facet with the specific geometry we just set up.
   ODFacet* prFacet = fSession->GetWindowState(ev)->
         CreateFacet(ev, rootFrame, clipshape, xtransform, kODNULL,
         kODNULL);
   xtransform->Release(ev);
   clipshape->Release(ev);

   // Set up a canvas based on the print job's port.
   ODCanvas* prCanvas = prFacet->CreateCanvas(ev, kODQuickDraw,
         (GrafPtr)thePrPort, kODFalse, kODFalse);
   prCanvas->SetPlatformPrintJob(ev, kODQuickDraw, (GrafPtr)thePrPort);

   // Make it the canvas of the facet we created.
   prFacet->SetCanvas(ev, prCanvas);
   rootFrame->FacetAdded(ev, prFacet);

   // Return the facet to the main print routine.
   return prFacet;
}
```

use OpenDoc's drawing code to image the page. Once we have a facet set up, we need only force the facet to update to get all of the parts to image on the page.

Note that this particular setup routine doesn't handle a very important case: QuickDraw GX–based part handlers. To set these up correctly, we would replace steps 1 and 2 of the recipe above with the following:

1. Check for the existence of QuickDraw GX printing using Gestalt.
   a. If QuickDraw GX printing isn't present, set up a normal Color QuickDraw print job.
   b. If QuickDraw GX printing is present, set up a QuickDraw GX print job. Also, set up a Color QuickDraw graphics port, for use with the QuickDraw–to–QuickDraw GX translator.
2. Set up a canvas representing the printing job, ready for use by both QuickDraw and QuickDraw GX.
   a. Install the QuickDraw–to–QuickDraw GX translator in your Color QuickDraw graphics port.
   b. Install the Color QuickDraw graphics port as the QuickDraw platform canvas in the canvas object.
   c. Install a view port that embedded part handlers can use as the QuickDraw GX platform canvas in the canvas object.

By setting up translators, a QuickDraw GX–based part handler can still work with embedded QuickDraw-based parts. This is a fairly complex (though straightforward) bit of code, one that we'll avoid for the purposes of this article. There's a complete recipe in the OpenDoc Software Development Kit for handling this case.

The next step is to loop through the pages, drawing each one. We'll assume that you can handle the details of creating a standard print loop yourself. Listing 9 shows the correct code to call in the middle of the page loop. This corresponds to steps 4a and 4b of our printing recipe. This routine gets the basic page geometry, resets the offsets and clipping for the page, and then forces the parts visible on the page to draw by calling an update on the root facet we've created elsewhere.

Listing 9. Printing a page

```
void MyPart::PrintPage(Environment *ev, ODFacet* prFacet, ODUShort page,
   ODRect *pageRect)
{
   // Get some basic printing geometry.
   ODRect   bbox;
   Rect     frect, qdPRect;

   ODShape* frameShape = prFacet->GetFrame(ev)->
         GetFrameShape(ev, kODNULL);
   frameShape->GetBoundingBox(ev, &bbox);
   bbox.AsQDRect(frect);
   frameShape->Release(ev);
   Point pt = {0,0};
   ODUShort locator = page-1;

   // Pick an appropriate offset, based on page number.
   pageRect->AsQDRect(qdPRect);
   while (locator) {
      pt.v += (qdPRect.bottom+1);
      locator--;
      if (PtInRect(pt, &frect))
         continue;
      else {
         pt.v = 0; pt.h += (qdPRect.right+1);
      }
   }

   // Make a transform for that offset.
   ODTransform* xtransform = prFacet->CreateTransform(ev);
   xtransform->SetQDOffset(ev, &pt);

   // Create a clip shape for the page, based on the transform.
   ODShape* clipshape = prFacet->CreateShape(ev);
   clipshape->SetRectangle(ev, pageRect);
   ODShape* invalshape = clipshape->Copy(ev);
   clipshape->Transform(ev, xtransform);
   xtransform->Invert(ev);
```

(continued on next page)

Listing 9. Printing a page *(continued)*

```
    // Change the geometry of the printing facet.
    prFacet->ChangeGeometry(ev, clipshape, xtransform, kODNULL);

    // Draw everything on the page. OpenDoc will call the Draw method
    // on every part visible on the page.
    prFacet->Update(ev, invalshape, kODNULL);

    // Clean up.
    clipshape->Release(ev);
    invalshape->Release(ev);
    xtransform->Release(ev);
}
```

Listing 10. Cleanup after printing

```
void MyPart::EndPrinting(Environment *ev, ODFacet* prFacet)
{
    // Find the printing canvas and facet; delete them.
    ODCanvas* prCanvas = prFacet->GetCanvas(ev);
    prFacet->GetFrame(ev)->FacetRemoved(ev, prFacet);
    delete prCanvas;
    delete prFacet;
}
```

Once the print loop is complete, we'll want to clean up. Listing 10 shows the routine that corresponds to step 5 of our printing recipe. It simply tosses away the facet and the canvas we created in Listing 8.

If you're performing more advanced printing, you may want to allow embedded parts to lay themselves out differently based on whether they're printing to a static or a dynamic canvas. In this case, you'll need to create new frames and perform layout negotiation. You should create nonpersistent frames, or they'll needlessly be written out to the document the next time the user saves. You'll probably want to do this:

1. Create a new nonpersistent frame for your part (the "master frame").
2. Create new nonpersistent frames for each embedded part and allow shape negotiation to occur on these new embedded frames.
3. Use the previous recipe to print the master frame.

ONWARD AND UPWARD

For simple cases, as you can see, the OpenDoc layout and graphics model isn't much more complex than what you probably already do. Even better, most of your code will work fine in the new model with just some additional setup and cleanup code. This is no accident, as OpenDoc has been designed from the start to allow you to reuse large parts of your code. So go ahead and show us some cool new parts with great graphics.

Thanks to our technical reviewers Jens Alfke, Steve Smith, and Joshua Susser.•

BALANCE OF POWER

Introducing PowerPC Assembly Language

DAVE EVANS

So far I've avoided the subject of PowerPC™ assembly language in this column, for fear of being struck down by the portability gods. But I also realize that a column on PowerPC development without a discussion of this subject would be too pious. Although today's compiler technology makes assembly language generally unnecessary, you might find it useful for critical subroutines or program bottlenecks. In this column I'll try to give you enough information to satisfy that occasional need.

If the thought of using assembly language still troubles you, please consider this as useful information for debugging. Eventually you'll need to read PowerPC assembly for tracing through code that was optimized, or when symbolic debugging just isn't practical. Also in this column, I'll cover the runtime basics that will help you recognize stack frames and routine calls during debugging.

USING POWERPC ASSEMBLY LANGUAGE

Assembly language on the PowerPC processor should be used only for the most performance-critical code — that is, when that last 5% performance improvement is worth the extra effort. This code typically consists of tight loops or routines that are very frequently used.

After you've carefully profiled your code and found a bottleneck routine in which your application spends most of its time, then what do you do? First you need an assembler; I recommend Apple's PPCAsm (part of MPW Pro or E.T.O., both available from APDA).

Next, you'll need to understand the instruction set and syntax. This column will give you a basic summary, but for a thorough reference you'll need the *PowerPC 601 RISC Microprocessor User's Manual*; to order one, call 1-800-POWERPC (1-800-769-3772).

Finally, you need to know the basic PowerPC runtime details — for example, that parameters are passed in general registers R3 through R10, that the stack frame is set up by the callee, and so on.

Once you have these tools and information, you can easily write a subroutine in assembly language that's callable from any high-level language. Then you'll need to review your code with the persistence of Hercules, fixing pipeline stalls and otherwise improving your performance.

THE INSTRUCTION SET AT A GLANCE

Many people think RISC processors have fewer instructions than CISC processors. What's truer is that each RISC instruction has reduced complexity, especially in memory addressing, but there are often many more instructions than in a CISC instruction set. You'll be amazed at the number and variation of the instructions in the PowerPC instruction set. The basic categories are similar to 680x0 assembly language:

- integer arithmetic and logical instructions
- instructions to load and store data
- compare and branch instructions
- floating-point instructions
- processor state instructions

We'll go over the first three categories here; you can read more about the last two in the PowerPC user's manual. Once you're familiar with the PowerPC mnemonics, you'll notice the similarity with any other instruction set. But first let's look at some key differences from 680x0 assembly: register usage, memory addressing, and branching.

KEY DIFFERENCES

Most PowerPC instructions take three registers as opposed to two, and in the reverse order compared to 680x0 instructions. For example, the following instruction adds the contents of register R4 and R5 and puts the result in register R6:

```
add    r6,r4,r5      ; r6 = r4 + r5
```

DAVE EVANS came to California in 1991 in search of temperate weather, having left Boston, the land of erratic and extreme climate. While in Boston he developed Macintosh software for a radical startup company and studied applied math at the Massachusetts Institute of Technology. At Apple, Dave has attended an estimated 1000 meetings, but in between them he managed to develop the Drag and Drop Developer's Kit. Dave is also trying to teach his pet iguana Herman to roll over, but without much success. •

Note that the result is placed in the first register listed; registers R4 and R5 aren't affected. Most instructions operate on the last two registers and place the result in the first register listed.

Unlike the 680x0 processors, the PowerPC processor doesn't allow many instructions to deal directly with memory. Most instructions take only registers as arguments. The branch, load, and store instructions are the only ones with ways of effectively addressing memory.

- The branch instructions use three addressing modes: immediate, link register indirect, and count register indirect. The first includes relative and absolute addresses, while the other two let you load the link or count register and use it as a target address. (The link and count registers are special-purpose registers used just for branching.) Using the link register is also how you return from a subroutine call, as I'll demonstrate in a moment.
- Load and store instructions have three addressing modes: register indirect, which uses a register as the effective address; register indirect with index, which uses the addition of two registers as the effective address; and register indirect with immediate index, which adds a constant offset to a register for the effective address. I'll show examples of these later.

The more complicated 680x0 addressing modes do not have equivalents in PowerPC assembly language.

On 680x0 processors, there are branch instructions and separate jump (**jmp**), jump to subroutine (**jsr**), and return from subroutine (**rts**) instructions. But in PowerPC assembly there are only branches. All branches can be conditional or nonconditional; they all have the same addressing modes, and they can choose to store the next instruction's address in the link register. This last point is how subroutine calls are made and then returned from. A call to a subroutine uses a branch with link (**bl**) instruction, which loads the link register with the next instruction and then jumps to the effective address. To return from the subroutine, you use the branch to link register (**blr**) instruction to jump to the previous code path. For example:

```
      bl    BB          ; branch to "BB"
AA:   cmpi  cr5,r4,0    ; is r4 zero?
      ...
BB:   addi  r4,r3,-24   ; r4 = r3 - 24
      blr               ; return to "AA"
```

Since conditional branches can also use the link or count register, you can have conditional return statements like this:

```
      bgtlr   cr5       ; return if cr5 has
                        ; greater than bit set
```

The instructions blr and bgtlr are simplified mnemonics for the less attractive **bclr 20,0** and **bclr 12,[CRn+]1** instructions. The PowerPC user's manual lists these as easier-to-read alternatives to entering the specific bit fields of the **bclr** instruction, and PPCAsm supports these mnemonics. But when debugging you may see the less attractive versions in disassemblies. •

ARITHMETIC AND LOGICAL INSTRUCTIONS

You've already seen the **add** and **addi** instructions, but let's go over one key variation before looking at other integer arithmetic and logical instructions. Notice the period character "." in the following instruction:

```
add.    rD,rA,rB    ; rD = rA + rB, set cr0
```

You can append a period to most integer instructions. This character causes bits in the CR0 condition register field to be set based on how the result compares to 0; you can later use CR0 in a conditional branch. In 680x0 assembly language, this is implied in most moves to a data register; however, PowerPC assembly instructions that move data to a register must explicitly use the period.

Other basic integer instructions include the following:

```
subf    rD,rA,rB    ; subtract from
                    ; rD = rB - rA
subfi   rD,rA,val   ; subtract from immediate
                    ; rD = val - rA
neg     rD,rA       ; negate
                    ; rD = -rA
mullw   rD,rA,rB    ; multiply low word
                    ; rD = [low 32 bits] rA*rB
mulhw   rD,rA,rB    ; multiply high word
                    ; rD = [high 32 bits] rA*rB
divw    rD,rA,rB    ; divide word
                    ; rD = rA / rB
divwu   rD,rA,rB    ; divide unsigned word
                    ; rD = rA / rB [unsigned]
and     rD,rA,rB    ; logical AND
                    ; rD = rA AND rB
or      rD,rA,rB    ; logical OR
                    ; rD = rA OR rB
nand    rD,rA,rB    ; logical NAND
                    ; rD = rA NAND rB
srw     rD,rS,rB    ; shift right word
                    ; rD = (rS >> rB)
srawi   rD,rS,SH    ; algebraic shift right
                    ; word immediate
                    ; rD = (rS >> SH)
```

Another flexible and powerful set of instructions is the rotate instructions. They allow you to perform a number of register operations besides just rotation, including masking, bit insertions, clearing specific bits, extracting bits, and combinations of these. Each rotate instruction takes a source register, a destination, an amount to shift either in a register or as immediate data, and a mask begin (MB) and mask end (ME) value. The mask is either ANDed with the result or is used to determine which bits to copy into the destination register. The mask is a 32-bit value with all bits between location MB and ME set to 1 and all other bits set to 0. For example, the following instruction will take the contents of R3, rotate it left by 5, AND it with the bit pattern 00001111 11111100 00000000 00000000, and place the result in register R4.

```
rlwinm    r4,r3,5,4,13   ; rotate left word
                         ; immediate, AND with mask
                         ; r4 = (r3 << 5) & 0FFC0000
```

Note that some assemblers allow you to specify a constant instead of the MB and ME values.

MOVING DATA

Getting data to and from memory requires the load and store instructions. There are a few variations, each with the addressing modes mentioned earlier. The amount of memory, the address alignment, and the specific processor will also affect how much time the operation will take. Here are some examples of specifying the size with load instructions:

```
lbz    rD,disp(rA)     ; load byte and zero
                       ; rD = byte at rA+disp
lhz    rD,disp(rA)     ; load half word and zero
                       ; rD = half word at rA+disp
lwz    rD,disp(rA)     ; load word and zero
                       ; rD = word at rA+disp
lwzx   rD,rA,rB        ; load word & zero indexed
                       ; rD = word at rA+rB
```

Note that the "z" means "zero," so if the amount loaded is smaller than the register, the remaining bits of the register are automatically zeroed. This is like an automatic extend instruction in 680x0 assembly language. You can also have the effective address register preincrement, by appending "u" for "update." For example,

```
lwzu   r3,4(r4)        ; r4 = r4 + 4 ; r3 = *(r4)
```

will first increment R4 by 4 and then load R3 with the word at address R4. The preincrement doesn't exist in 680x0 assembly, but it's similar to the predecrementing instruction **move.l d3,-(a4)**. There's also an option for indexed addressing modes — for example, "load word and zero with update indexed":

```
lwzux  r3,r4,r5        ; r4 = r4 + r5 ; r3 = *(r4)
```

This instruction will update register R4 to be R4 plus R5 and then load R3 with the word at address R4.

Store instructions have the same options as load instructions, but start with "st" instead of "l." (The "z" is omitted because there's no need to zero anything.) For example:

```
stb    rD,disp(rA)     ; store byte
sthx   rD,rA,rB        ; store half word indexed
stwux  rD,rA,rB        ; store word update indexed
```

A word of caution: Do not use the load or store string instructions (**lswi**, **lswx**, **stswi**, and **stswx**) or load multiple instruction (**lwm**). Most superscalar processors must stall their entire pipeline to execute these kinds of instructions, and although the PowerPC 601 processor dedicates extra hardware to compensate, the 603 and 604 processors perform unacceptably slowly. Loading each register individually will result in faster execution.

COMPARE AND BRANCH

A compare instruction operates on one of the eight condition register fields, CR0 to CR7. It compares a register against either another register or immediate data, and then sets the four condition bits in that condition register field accordingly. The bits are as follows:

bit 0	less than
bit 1	greater than
bit 2	equal to
bit 3	copy of summary overflow bit

If you're wondering how to test for greater than or equal to, you're paying attention: You can test whether each bit is true or false, so to test for greater than or equal to, just see if the less-than bit is false. The last bit is a copy of an overflow bit from the integer or floating-point exception register. For more information on exceptions, see the PowerPC user's manual.

The official mnemonics for compare instructions include a 64-bit option, but until PowerPC registers are 64-bit, the following simpler 32-bit mnemonics are used:

```
cmpwi     CRn,rA,val   ; compare word immediate
                       ; rA to val
cmpw      CRn,rA,rB    ; compare word
                       ; rA to rB
```

```
cmplwi    CRn,rA,val   ; compare logical word
                       ; rA to val (unsigned)
cmplw     CRn,rA,rB    ; compare logical word
                       ; rA to rB (unsigned)
```

The "w" stands for "word" and means these are the 32-bit compare instructions. The "l" means the comparison is logical and therefore unsigned.

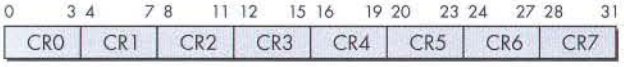
Now let's look at the branch instructions. We covered basic branch instructions earlier, but here are some examples of common simplified branch mnemonics:

```
bgt    CRn,addr     ; branch if CRn has greater
                    ; than bit set true
ble    CRn,addr     ; branch if CRn has greater
                    ; than bit set false (tests
                    ; for less than or equal)
bgtl   CRn,addr     ; set link register, branch if
                    ; CRn has greater than bit set
```

Also useful are the decrement counter conditional branches. They allow you to load the count register and, in one instruction, decrement it and branch based on its value and another condition. For example:

```
dbnz   addr         ; CTR = CTR - 1
                    ; branch if CTR is nonzero
dbz    addr         ; CTR = CTR - 1
                    ; branch if CTR is zero
dbzt   bit,addr     ; CTR = CTR - 1
                    ; branch if CTR is zero and
                    ; condition bit is set true
```

The **dbzt** instruction's bit testing brings up an important point. Conditional branches specify either a condition register field or a condition bit. As shown below, the condition register fields are placed side by side in a single 32-bit condition register. When a branch mnemonic requires a field, it needs a value from 0 to 7 to specify which 4-bit field to use. When a branch mnemonic requires a bit value, it needs a number from 0 to 32 specifying a bit in the whole condition register. Bit number 0 is the high (less than) bit in CR0, bit number 4 is the high bit in CR1, and so on. (Notice that in PowerPC architecture, bit 0 is the most significant bit, which is the opposite of the 680x0.)



Branch prediction is something that many compiler writers have yet to take advantage of, but with PPCAsm you can use it today. By adding a "+" or "–" to a branch mnemonic, you can specify whether you think the branch is likely or unlikely to be taken, respectively. For example:

```
bgt+   cr0,addr     ; predict branch taken
```

However, this works only if the target address is in the same source file. Branch prediction on the PowerPC 601 and 603 is determined by the target address of the branch — specifically, on whether the target address is before or after the branch instruction. So if the target routine is in another source file, the compiler can't determine if the target address will be before or after the branch instruction, and therefore can't set the branch prediction bit accurately. See the Balance of Power column in Issue 20 for more information on branch prediction.

CALLING CONVENTIONS

The PowerPC processors have 32 general-purpose registers, 8 condition register fields, and 32 floating-point registers. Just as in the 680x0 Macintosh run time, most registers are available for general use. But some are reserved for specific duties: general register R1 is the stack pointer, and R2 is the RTOC or Register for Table of Contents. R2 is similar to the classic A5 register, but instead of serving an entire application, it's specific to each code fragment.

Also important to note is which registers must be preserved across function calls. Registers R13 to R31, FPR14 to FPR31, and CR2 to CR4 must be saved and restored if you use them in your function. It's all right to store them in a scratch register if you don't call another subroutine. You can always use registers R3 through R10, for example, without any additional work.

Optimized code doesn't always use stack frames, and if your assembly is just for tight utility routines you won't need them. But if you call other subroutines, your routine must set up a frame. This will also aid in debugging. When your assembly routine is called, the stack pointer will point to the caller's stack frame. Your routine should set up a frame with space for local variables plus the standard frame size of 56. It should also save the return address in the frame and clean up before exiting. Here's the recommended code to do this:

```
mflr   r0                ; move return addr to r0
stw    r0,8(sp)          ; save r0 in stack frame
stwu   sp,-frame(sp)     ; set up new frame
...                      ; your code here
lwz    r0,frame(sp)+8    ; return address to r0
addic  sp,sp,frame       ; remove frame
mtlr   r0                ; restore return
blr                      ; return
```

The size of the frame is variable, but at a minimum is 56 bytes for parameter space and special register storage. If you save and restore any variables, or need local stack variables, add the size needed to 56. The frame size must be a multiple of 8, to leave the stack double-word aligned. Add padding to your frame to make sure it's a multiple of 8 bytes.

Subroutine calls within your code fragment use just a simple instruction-relative branch and link. If you call subroutines outside your fragment, such as into the Toolbox, you need to put a **no-op** instruction after that branch. The **no-op** is actually the impotent **ori r0,r0,0** instruction. The linker will replace this **no-op** with an instruction to restore your RTOC register after the call. It will also add special cross-TOC glue code and redirect the branch to that glue. This is necessary because you must set up the callee's RTOC so that it can access its globals, and your code is responsible later for restoring your RTOC.

Here's an example of this cross-TOC glue:

```
lwz    r12,routine(RTOC)  ; load t-vector
stw    RTOC,20(RTOC)      ; save my RTOC
lwzr   0,0(r12)           ; get callee address
lwz    RTOC,4(r12)        ; set callee RTOC
mtctr  r0                 ; prepare branch
bctr                      ; jump to callee
```

You'll often see this glue during low-level debugging. The first instruction gets a *transition vector* (or *t-vector*) from your global data and places it in R12. This vector is a structure containing the callee's address and RTOC, and it's filled in by the Code Fragment Manager when your code binds to the callee's fragment. Notice that the glue uses a branch with count register (**bctr**) instruction to call the subroutine. This uses the count register as a target address so that the link register with your return address will remain unmodified; therefore, don't make cross-TOC calls in loops that use the count register.

OPTIMIZING FOR SPEED

Let's look at a simple routine in C that compares two Pascal strings:

```
Boolean pstrcompare(StringPtr p1, StringPtr p2)
{
   short length, i;

   if ((length = p1[0]) != p2[0]) return false;
   for (i = 1; i <= length; ++i)
      if (p1[i] != p2[i]) return false;
   return true;
}
```

Compiling this with the PPCC compiler and using the optimizer for speed produces the assembly code shown below. (While it certainly is possible to tune the C code directly, we'll ignore that for the purposes of this example.)

```
      lbz      r11,0(r3)     ;  r11 = length p1
      lbz      r5,0(r4)      ;  r5 = length p2
      cmpw     cr0,r11,r5    ;· compare lengths
      beq      pre           ;·
      li       r3,0          ;  nope, return false
      blr
pre:  cmpwi    cr0,r11,1     ;  check length
      li       r12,1         ;  load count
loop: blt      pass          ;· done?
      lbzx     r5,r3,r12     ;  r5 = p1[i]
      lbzx     r6,r4,r12     ;  r6 = p2[i]
      cmpw     cr0,r5,r6     ;· equal?
      bne      fail
      addic    r5,r12,1      ;  add 1
      extsh    r12,r5        ;· extend and move
      cmpw     cr0,r11,r12   ;· check if done
      b        loop
pass: li       r3,1          ;  return true
      blr
fail: li       r3,0          ;  return false
      blr
```

Looking at this code, we notice that the two StringPtr parameters are passed in R3 and R4. The first six instructions check the lengths of these two strings and return false if they're not equal. Then the loop preloads a count and uses **cmpwi cr0,r11,1** to see if it needs to iterate even once. The loop is simple, but it does an extraneous **extsh** instruction because the optimizer doesn't realize R12 is already a full word.

The key to optimizing PowerPC assembly code is to keep the processor's pipeline from stalling. This isn't always possible, and different PowerPC processors have different pipelines, but you can usually arrange your assembly code for significant performance improvements on all PowerPC processors.

For more information on pipelines and different optimization techniques, see the article "Making the Leap to PowerPC" in *develop* Issue 16 and the Balance of Power column in Issues 18 and 19.•

The situations that most often stall the pipeline are memory access, register dependencies, and conditional branch instructions. If data is loaded from memory and then used immediately, you'll stall the pipeline at least one cycle and possibly more for cache or page misses. If one instruction writes to a register and the next instruction references the same register, the processor

might not be able to finish the second instruction until after the first one completes. The processor alleviates this by executing instructions out of order or with temporary registers, but you may nonetheless waste cycles. Also, if a branch is directly preceded by the needed comparison, the processor may mispredict the branch or just stall until the compare is done.

The key tactic for addressing these situations is to reorder your instructions. Move loads and stores as early in your code as possible, as they may take a long time to service. Then if two instructions reference the same register, find another unrelated instruction and move it in between. The same goes for conditional branch instructions: try to put as many other instructions between the compare and the branch as possible. As examples, look for the "·" characters in the above sample code; these denote possible pipeline stall points. Note, however, that the 603 and 604 microprocessors issue instructions differently such that you shouldn't bunch loads and stores together.

Other general tactics can improve your speed. Use as many scratch registers as possible and go to the stack for local storage only if you absolutely must. The same applies to your stack frame: only save to it things that will be modified in your routine. For example, if you don't call any subroutines, don't save your link register there. Loops should use the one-step decrement branch (**bdnz**) instruction.

Finally, read the PowerPC user's manual before going to bed every night for time-saving instructions like **rlwimi** (rotate left word immediate with mask insert).

Now let's optimize the above example by hand:

```
pstrcompare:
      lbz       r7,0(r3)      ;  r7 = length p1
      mr        r6,r3         ;  save a copy of p1
      lbz       r8,0(r4)      ;  r8 = length p2
      li        r3,0          ;  preload false
      addi      r5,r7,1       ;  add 1 for count
      mtctr     r5            ;· preload count
loop: cmpw      cr0,r7,r8     ;  equal?
      lbzu      r7,1(r6)      ;  r7 = *(++p1)
      bnelr                   ;  return if ¬ equal
      lbzu      r8,1(r4)      ;  r8 = *(++p2)
      bdnz      loop
pass: li        r3,1          ;  return true
      blr
```

Here we've removed all the key stall points by doing more work before the loop and also modifying the loop. With **lbzu** autoincrementing and **dbnz** autodecrementing instructions, the loop is now only five instructions long, compared to the earlier nine instructions and one stall point. To achieve this we also needed to preload R3 and the count register, but we did that additional work in stall points. The **mtctr** instruction can be expensive, with a latency of three or more cycles; however, using the count register reduces the work done within a loop, and that often makes up for the added **mtctr** cycles.

The earlier PPCC-optimized version would take about 110 cycles to verify that two 10-byte strings were identical. Our hand-tuned version takes only half as long. And although string comparisons are probably not your critical bottleneck, this same procedure can be applied to your critical code.

PROCEED WITH CAUTION

Any code you write in assembly language is not portable and is usually harder to maintain. You also don't get the type checking and warnings that a compiler provides. But for code that must be faster than the competition, you may want to hand-tune in PowerPC assembly language.

One *strong* word of caution: Do not use IBM POWER instructions! They may work on the 601 processor, which supports them, but they will not run on any other PowerPC processor. If you use them, your software may crash or run significantly slower on future Power Macintosh models. To make sure your code is clean of POWER instructions, you can use Apple's DumpXCOFF or DumpPEF tool, both of which have an option to check for invalid instructions. There's also a list of POWER instructions supported by the 601 in Appendix B of the PowerPC 601 user's manual.

Another warning: Most instructions take registers, immediate data, or bit numbers as arguments, and the assembler will assume you're setting them correctly. It's easy to think you've specified a bit number but in fact have used a critical register by accident. These bugs are hard to find. Our earlier **rlwinm** example can be written **rlwinm 4,3,5,4,13**; it's easy to see how argument meanings can be confused. You might try the **-typecheck** option of PPCAsm version 1.1 to help catch mistakes, but please be careful!

Thanks to Dave Falkenburg, Tim Maroney, Mike Neil, and Andy Nicholas for reviewing this column.•

A First Look at Dylan: Classes, Functions,and Modules

Dylan is a new object-oriented dynamic language (OODL) that's attracting a lot of attention. Like C++, it's designed for efficient compilation and delivery of mainstream commercial applications. However, Dylan differs from C++ in important ways that make it more powerful and flexible. Here we'll focus on one important difference from C++: the way classes and their methods are organized.

STEVE STRASSMANN

The organization of classes and functions is different in Dylan than in C++. In C++, classes are used in two ways: to encapsulate data and as a scoping mechanism. Methods in C++ "belong to" classes, and there are many complex mechanisms governing access to methods from "inside" and "outside" a class. In Dylan, classes are used only for data encapsulation — there's no notion of methods being owned by classes. As a result, specifying and using methods is cleaner, simpler, and more expressive.

Access is simplified and abstracted through *modules*, which are a way of grouping related classes, methods, and variables. Rather than being tied to a single class, each method belongs to a family called a *generic function*. Each generic function can operate on one or more related classes, and can be extended across one or more modules. We'll talk more about generic functions, polymorphism, and modules later in this article.

Dylan has many other features that distinguish it from C++, including:

- automatic memory management
- clean, consistent syntax
- fully and consistently object-oriented model
- dynamic as well as static type checking
- support for incremental compilation
- first-class functions and classes

There's not enough space here to do justice to each of these topics, so we'll just touch on some of them as we discuss classes, functions, and modules. As you might expect, this article assumes you have some familiarity with basic object-oriented concepts, such as classes, instances, and inheritance.

On this issue's CD, you'll find a freeware Dylan interpreter, called Marlais, that you can use to execute code written in Dylan. Simply run the application and enter your

STEVE STRASSMANN (AppleLink STRAZ, Internet straz@apple.com) has a patent on surgical catheters (#4,838,859) and is the co-author of the infamous *UNIX-Haters Handbook*. After getting his Ph.D. at the MIT Media Lab in entertainment engineering, he joined the Dylan team at Apple in Cambridge, Mass.•

code at the prompt. Also on the CD are the code samples you'll see in this article, plus the *Dylan Interim Reference Manual* and other Dylan goodies.

Apple's implementation of Dylan, called Apple Dylan, is planned to ship later this year. One great feature of Apple Dylan is that it allows you to call existing C and C-compatible code libraries, such as the Macintosh Toolbox. See "Creole: Using the Toolbox and Other C Code From Within Dylan Code" for details.

CREOLE: USING THE TOOLBOX AND OTHER C CODE FROM WITHIN DYLAN CODE

With any new language, you're bound to wonder whether you'll be able to get at the "really good stuff." You know, interfaces always seem to be published just for C programmers, and nobody else. I don't mean merely the Macintosh Toolbox, but any other code already written by you or a third party, like database access routines or advanced graphics libraries. In many cases (such as with the Macintosh Toolbox), you may not have access to the source code, so recompiling or translating it into the new language is simply not an option.

Apple has designed a cross-language extension to the Dylan language. This extension, called Creole in Apple Dylan, allows you to build programs with parts written in both Dylan and C or C-compatible languages. We at Apple hope the extension will be supported by other Dylan implementations, but since the extension isn't part of the standard Dylan language, it's not required. (The Marlais interpreter on this issue's CD doesn't support it.) In the future, Apple will also support the System Object Model (SOM) extension, which is used by OpenDoc. Here we'll take a look at some features of Apple Dylan's Creole implementation.

Once you import C interfaces into Dylan, you can call C functions and refer to C **struct**s as if they were Dylan functions and objects. There's no need to translate the C headers first; Creole reads them directly. In the following simple example, we import the interface file OSUtils.h, which contains the Toolbox function SysBeep; we can then, for instance, call **SysBeep(1)** from Dylan.

```
define interface
   #include "OSUtils.h",
      import: {"SysBeep"};
end interface;
```

Creole provides these additional facilities:

- An access path (linking) mechanism links compiled C-compatible modules, including C++, Pascal, assembler, and FORTRAN modules, into a Dylan application. Creole supports object (".o") files, shared libraries (Apple Shared Library Manager or Code Fragment Manager), inline traps, code resources, and PowerPC transition vectors.
- Cross-language calls allow Dylan routines to call routines in another language, and vice versa.
- Name mapping translates names of entities in another language into Dylan variable names in a specified module. Apple Dylan offers several convenient mappings for common naming conventions.
- Type mapping translates C types into Dylan types and provides type checking for Dylan clients of the Macintosh Toolbox and other interfaces.
- Low-level facilities provide Dylan programs with direct use of machine pointers and the raw bits pointed to by the machine pointers.

A **define interface** statement imports one or more C interface files and creates Dylan classes, constants, and functions corresponding to the C types, constants, and functions in the interface files. Like any Dylan expression, a **define interface** statement exists in a particular module, as do the variables that it defines. You can export and rename these variables using module options just as you would for normal Dylan variables (as discussed later under "The Role of Modules").

Many options are available to override Creole's default behavior. For example, you can do any of the following:

- selectively import parts of an interface
- explicitly control type mapping — for example, to map **StringPtr** to **<Pascal-string>**
- explicitly control name mapping to avoid name conflicts because of the difference in case-sensitivity and scoping rules in Dylan and C
- work around undesirable features in the interface or in Creole
- control tradeoffs between runtime memory consumption and dynamic functionality

CLASSES AND OBJECTS

Dylan is fully and consistently object-oriented, much like Smalltalk™. Everything is an object, including numbers, strings, and even functions and classes themselves. Each object descends from a single common ancestor class, named **<object>**.

> **The <> characters** are not some fancy operator but are merely a typographic convention for indicating the name of a class in Dylan, just as all-uppercase letters might indicate a macro in C++. You're allowed to name a class without the **<>** characters, but that would be considered bad style.•

To illustrate how classes are used in Dylan, let's look at one of our samples, SimMogul, to model Hollywood high finance. In Listing 1, we define a few classes, creating the inheritance hierarchy shown in Figure 1.

Listing 1. SimMogul — basic version

```
define class <project> (<object>)
   slot script;                    // All you need is a hot script
   slot star;                      //  and a big name.
end class <project>;               // Last two words are optional.

define class <actor> (<object>)
   slot name;                      // Actor's name
   slot salary;                    // Cost to hire
   slot fans;                      // Audience size
end class;

define class <script> (<object>)
   slot name;                      // Script's name
   slot fx-budget;                 // Cost of special effects
end class;

define class <sci-fi> (<script>)
end class;

define class <romance> (<script>)
end class;
```

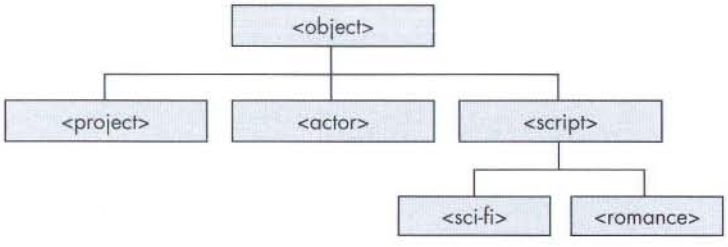


Figure 1. Inheritance hierarchy of SimMogul classes

The first thing you might notice about this code is that Dylan identifiers draw on a richer stock of characters than do most languages. Dylan identifiers are case-insensitive and can include characters like <, >, *, +, and –, which are traditionally

reserved for operators. As a result, operators like these must be surrounded by spaces when used in formulas (as you'll see later in the definitions for **profits** in Listing 5).

As shown in Listing 1, each **define class** statement begins with the name of the class being defined, followed by its parent (or *superclass*) in parentheses. Dylan supports multiple inheritance; multiple superclasses would be listed in the parentheses, separated by commas. For this short example, however, we'll stick to single inheritance.

<project> is a simple class with two *slots* (comparable to data members in C++) named **script** and **star**. This is a pretty basic structure that doesn't include any options, but it illustrates the syntax for class definitions. There's no need to create constructor or destructor methods; that's taken care of automatically. The last two words, **class <project>**, are optional, but if you provide them, the name must match that of the class being defined. You can just say **end** or **end class** instead if you like, which is what we've done for the remaining classes.

TYPE DECLARATIONS AND AUTOMATIC MEMORY MANAGEMENT

Type declarations are optional in Dylan because values, not storage locations, are strongly typed. Each object's type is always known from the moment it's created, so there's less need to declare types on storage locations. It's OK to leave off type declarations, as we did for slots in Listing 1. This makes rapid prototyping much easier than in C++.

Listing 2 shows a version of SimMogul that does contain some type declarations. The definition of **<actor>**, for example, has a slot declared as **name :: <string>**, which specifies the type of the **name** slot. The compiler will generate code that guarantees that only strings can be stored in this slot; attempts to store anything else will cause an error.

Another reason to provide type declarations is that it allows the compiler to generate more efficient code. For example, if you wrote code that stores an appropriately declared value in an **<actor>**'s **name** slot, at compile time the compiler would be able to deduce the value's type. Values that are known to be strings will be stored efficiently,

Listing 2. SimMogul — embellished version

```
define class <project> (<object>)
   slot script;                    // All you need is a hot script
   slot star;                      //  and a big name.
end class <project>;               // Last two words are optional.

define class <actor> (<object>)
   slot name :: <string>,          // Actor's name
      required-init-keyword: name:;
   slot salary :: <number>,        // Cost to hire
      init-value: 1000000,
      init-keyword: salary:;
   slot fans :: <number>,          // Audience size
      init-value: 1000000,
      init-keyword: fans:;
end class;
```

(continued on next page)

Listing 2. SimMogul — embellished version *(continued)*

```
define class <script> (<object>)
   slot name :: <string>,          // Script's name
      required-init-keyword: name:;
   slot fx-budget :: <number>,     // Cost of special effects
      init-value: 10000,
      init-keyword: fx-budget:;
end class;

define class <sci-fi> (<script>)
   inherited slot fx-budget, init-value: 20000000;
end class;

define class <romance> (<script>)
   inherited slot fx-budget, init-value: 0;
end class;

define variable arnold =
   make(<actor>, name: "Arnold", fans: 10000000);

define variable betty =
   make(<actor>, name: "Betty", fans: 5000000);

define variable tender :: <script> =
   make(<romance>, name: "Tender Sunshine");

define variable zarx :: <script> =
   make(<sci-fi>, name: "Land of the Zarx-Eaters");

define constant $ticket-price = 7;
```

with no runtime type checking. Those known not to be strings will generate compile-time warnings, just as they would in a strongly typed language. If you choose to leave off declarations, the compiler will insert instructions for runtime type checking, so you'll have crash-safe code no matter what. This is an example of how Dylan always lets you compile in a way that's both maximally safe and efficient.

In general, Dylan programs should crash much less often than comparable C programs, because most errors will be detected and handled gracefully and automatically. Automatic memory management is one big source of this safety, since it eliminates the majority of bugs that usually come from manually operating on raw memory pointers. Dylan's ability to ensure safety, however, is limited when working with code imported from outside Dylan, such as the Macintosh Toolbox, which forces Dylan programmers to use raw memory pointers in some cases. Apple Dylan will insulate programmers as much as possible from these pointers with an application framework.

CREATING OBJECTS AND FILLING THEIR SLOTS

Your application creates objects by calling **make**, which creates *instances* of a class. Listing 2 gives some examples of actors and scripts being created with calls to **make**. Values for slots are provided with keyword arguments to **make**, called *init-keywords*. Dylan keywords, which are similar to Smalltalk keywords, are a way to provide

optional function parameters. I'll have more to say about specifying and using keywords in function calls in the section on functions.

Since the slots in **<project>** don't have init-keywords, you can't provide values for them when you use **make** to make projects. If a project is created with **make(<project>)**, the slots are uninitialized, and any attempt to read their values in this uninitialized state is an error that will be detected and reported.

The **name** slot in **<actor>** has a **required-init-keyword:** option, which is used further down to specify the name of the **arnold** object. Required init-keywords are commonly used for slots with no default value because this requires callers to provide a value when they make objects.

The other slots in **<actor>**, **salary** and **fans**, have default values as well as init-keywords. When an actor is created, the slot's value can be defaulted (for example, **arnold**'s **salary**) or overridden (for example, **arnold**'s **fans**). Slots can also be initialized with the **init-function:** option, which calls a function to compute the default value.

The declaration **salary :: <number>** restricts the **salary** slot to hold only numbers. Notice that we didn't choose a specific numeric type for the **salary** slot type (such as short, int, long, or double), though we easily could have. Dylan provides a rich library of numeric types, including integers of unlimited size (which are good for devalued currencies and salaries of major athletes). By using **<number>** instead of a more specific numeric type, your type declaration becomes a tool for documentation and error checking, even while you're in the midst of rapid prototyping. We're not obliged to make some arbitrary and premature optimization at this stage, as we would with C or C++. Using **<number>** captures as much of our design as we want for now; we can always come back and tune it later.

A Dylan class inherits slots from all its superclasses and can also define its own new slots, just as in C++. All slots in a given class must be unique; there cannot be two different slots with the same name. You can override some properties of an inherited slot, however, by partially respecifying the slot. Taking a look at the definition of **<sci-fi>** in Listing 2, we see that it overrides the default init-value for **fx-budget** inherited from **<script>** with a somewhat higher value. The keyword **inherited** indicates that the slot is inherited from a superclass; it's not a new slot with the coincidentally identical (and therefore illegal) name.

You can specify many other interesting options for slots, such as **class** allocation, which shares a singly allocated value used by all instances of that class; **class** allocation roughly corresponds to a static data member in C++. Dylan also lets you provide **virtual** allocation for slots. Rather than being stored in the slot, a virtual slot's value is computed by a function each time the slot is referenced. This feature is missing from C++ and is very different from what C++ refers to as virtual data members.

USING VARIABLES AND CONSTANTS

In Listing 2, we make some objects out of the classes and bind them to global variables with the **define variable** statement. The variables holding the actors have no type declaration — we didn't do this with any design considerations in mind, but just to show you that it can be done. Like slot declarations, type declarations for global variables are optional; they're used to increase efficiency, not to change the program logic. The other two variables have **:: <script>** type declarations, which is OK, since the values stored there are indirect instances of **<script>**. The variable **tender** is an instance of **<romance>**, which is a subclass of **<script>**.

Also included is a **define constant** statement, which looks just like **define variable**, except that once you give it a value, the running program isn't allowed to change it. The **$** in the name **$ticket-price** is something of a coincidence. By convention, all constants in Dylan are given names beginning with a dollar character, as in **define constant $pi = 3.14159**.

It's worth noting that **define constant** doesn't restrict mutable objects from being mutated. Some collections, such as vectors, are mutable in that the value of an element can change, and class instances are mutable in that a slot can change (unless you declare the slot as a **constant** in the class definition, of course). Since **define constant** describes the identifier, not the object, what it really means is that the identifier will always refer to *that particular object*, and to no other object. This is the same as a **const** pointer in C++, where the pointer is not allowed to change but the object pointed to may be mutated.

$ticket-price is a real constant after all, because its value of 7 (like all numbers) cannot be mutated; for example, you cannot change the 7 to an 8 without changing the object itself.

VARIABLES HOLD ANYTHING

Variables (and constants, which are a kind of variable) can contain any type of Dylan data object, including numbers, strings, and user-defined objects like actors and scripts. But in Dylan, the classes and functions themselves are also objects, and hence are also stored in variables. It turns out that **<actor>** is just another variable, as is **arnold**. The value of the variable whose name is **<actor>** happens to be a class, and the value of the variable whose name is **arnold** is an instance of that class.

When we say everything's an object in Dylan, we mean *everything*. A variable is just a way of naming an object so that you can refer to it in your program. Since you can refer to functions or classes just as easily as you can refer to numbers, we think of them as "variables." So don't be shocked when you see documentation referring to something like **print** as a variable. It's just a variable whose value happens to be a function.

HOW FUNCTIONS WORK IN DYLAN

Dylan uses a simple, consistent, functional interface for slot access, which avoids many of the confusing aspects of C++'s data members. Functions in Dylan have many elegant features that make them more powerful than their counterparts in C++, but without adding a lot of complicated syntax. In this section we'll talk about some of the ways you can create and use Dylan functions.

GETTERS AND SETTERS

By default, a pair of accessor functions, called *getter* and *setter* functions, is created for each slot. For example, the definition of **<actor>** in Listing 1 automatically creates the following six functions:

```
name(a)                  // Gets the name of actor a
name-setter(new, a)      // Sets the name of actor a to new
salary(a)                // Gets the salary of actor a
salary-setter(new, a)    // Sets the salary of actor a to new
fans(a)                  // Gets the audience size of actor a
fans-setter(new, a)      // Sets the audience size of actor a to new
```

Slot access in Dylan looks exactly like a function call, even though the compiler may implement slot access much more efficiently. Alternatively, you can use the more

traditional dot notation for slot access. Therefore, the syntax **object.property** is exactly equivalent to **property(object)**. You can use whichever syntax best fits the situation.

This functional interface is a great feature, because it allows a class's implementation details to remain an abstraction for the users of a class. The **fans** property, which indicates the box office drawing power, might be stored as a slot in some classes or it might be computed on the fly by a function for other classes. Users will always see a functional interface, and never need to know about the internal implementation.

Whenever a slot reference appears on the left side of an assignment statement, the reference is translated into a call to the appropriate setter function. For example, these are all equivalent ways of changing the **name** slot of the **arnold** object:

```
arnold.name := "Arnie";
name(arnold) := "Arnie";
name-setter("Arnie", arnold);
```

Slots can also take a **setter:** option, which lets you provide the name of the setter function. The default is to give it a name like **name-setter**, but you can use a different name, or specify that no setter at all should be created. If there is no setter function, you effectively make the slot's value read-only. As you'll see later in the section on modules, you can also control read and write access to slots by selectively exporting getter and setter functions to other modules.

POLYMORPHISM

Object-oriented languages, including Dylan, provide *polymorphic* functions, which means a given function may be executed as one of several possible implementations of that function, called *methods*. In our code above, **name** is just such a function. Calling **name(arnold)** calls the **name** method for actors, but calling **name(tender)** invokes the **name** method for scripts, which may have a very different implementation.

So, when Dylan sees a call to **name(x)**, depending on what type of object **x** is, one of several methods is selected and executed. In Dylan, **name** is called a *generic function*, consisting of a family of **name** methods that implement the functionality of **name** for various classes (see Figure 2). Each of these methods "belongs to" its generic function (in this case, **name**) rather than to a class. This is a key point; it's the core difference between C++'s object model and Dylan's.

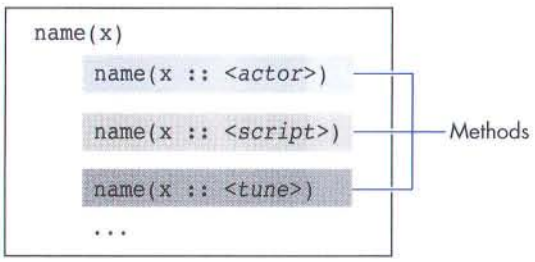


Figure 2. Generic function containing several methods

C++'s virtual methods are polymorphic only to the extent that they share a common ancestor. In C++, if you wanted **name** to work on both actors and scripts, you'd have to create a class (for example, **nameableObject**) just to contain the **name** method, and then modify both actor and script classes to inherit from it. This scenario creates quite a few unwanted complications. First, it clutters up your object hierarchies with

unnatural "glue" classes that have little to do with the problem domain being represented. Second, it requires you to add inheritance links to bring together classes that otherwise have no reason to be connected, which reduces modularity. Multiple inheritance is extremely awkward in C++ (much less so in Dylan), so you usually want to avoid it wherever possible when using C++.

You also may not have the desire or the ability to change classes near the root of a C++ class hierarchy, either because you don't have access to the affected classes' source code, or because the recompilation time would be very costly. The latter is usually not a problem in Dylan, because most commercial Dylan implementations (including Apple Dylan) provide incremental compilation, which means you can edit, recompile, and relink classes in a matter of seconds.

METHOD SPECIFICITY

As with slots and global variables, type declarations for Dylan function parameters are optional. Providing type declarations, which is called *specializing* the method, restricts a method to be valid for a specific set of operands. Listing 3 shows several methods belonging to the **double** generic function, specialized for various parameter types. (The value returned by a Dylan function is simply the value returned by the last expression executed in its body; there's no need for an explicit **return** statement.)

Listing 3. Method specificity

```
define method double (x)      // No type declaration, default handler
   pair(x, x);                // Works on any type of object
end method double;

define method double (x :: <number>)     // Works on all numbers
   2 * x;                                // Returns 2 times x
end method double;

define method double (x :: <string>)     // Works on all strings
   concatenate(x, x);                    // Returns "stringstring"
end method double;

define method double (x == cup:)         // Works on the symbol cup:
   pint:;                                // Returns the symbol pint:
end method double;
```

The first method in Listing 3 has no specialization at all, so it's equivalent to a default specialization of **x :: <object>**, which means it will work on anything. It returns a new structure (an instance of the built-in class **<pair>**), containing two pointers to the argument **x**.

The default specialization might not be satisfactory for all objects, of course, so the second method specializes the behavior for the case where **x** is a **<number>**. In this case, **double** returns the argument multiplied by 2. For a **<string>**, the third method returns a new string created by concatenating the argument to itself.

Dylan provides a large library of collection types, including strings, vectors, hash tables, and much more, along with an extensive and highly consistent library of operations on them. Working with Dylan's collections is much easier than with C, since you don't have the administrative headaches of manual storage management.•

The last method in Listing 3 illustrates Dylan's ability to specialize on specific instances (called *singletons*), not just whole classes. Through the use of **==** rather than **::**, the parameter is constrained by an equality test, not class membership. The object in this case is a symbol, which is an interesting data type not found in C or C++. A symbol is a case-insensitive immutable string, often used where you might use an **enum** in C. In this method, **double** is defined to return the symbol **pint:** whenever the argument is the symbol **cup:**.

> **The foo: syntax** is a convenient way to refer to symbols in your code, but it can be confusing, especially when passing symbols as keyword parameters in function calls. Dylan provides a second, equivalent syntax for symbols, which looks like a string with a **#** (for example, **#"foo"**). This also lets you create symbols with spaces in their names.•

When **double** is invoked on an argument, the most specific method is invoked. Singletons are considered to be the most specific; if a match isn't found, a method for the most specific matching parameter type is found. For example, **double("foo")** would invoke the third method, because **<string>** is more specific than **<object>**, which is what the first method is specialized to. If no match is found, Dylan will catch it and signal an error.

OTHER PARAMETER TRICKS: #REST, #KEY, AND RETURNED VALUES

In addition to having the normal kind of parameters (also called *required parameters*), whose number and position are fixed, Dylan functions can take varying numbers of additional parameters.

A **#rest** parameter collects an arbitrary number of arguments as a sequence. For example, the following function takes one required argument, **view**, and any number of additional arguments. A **for** loop is used inside the function to iterate over the arguments.

```
define method polygon (view :: <view>, #rest points)
   for (p in points)
      ...
   end for;
end method;
```

Here's an example of using this function:

```
polygon(myWindow, p1, p2, p3, p4, p5);      // Typical usage
```

Keyword parameters specified with **#key** are quite handy, especially for functions with many parameters, which often take default values. As we saw earlier, **make** takes keyword parameters in order to create objects. These can be provided in any order by the caller, or omitted entirely if default values are specified. The keywords themselves provide an extra degree of clarity to the calling code, since they serve to document the arguments they introduce. For example:

```
define method rent-car (customer :: <person>,  // Two required parameters
                        location :: <city>,     //  and up to 4 keywords
                        #key color = white:,    // Default color is white
                           sunroof? = #f,       // Default no sunroof
                           automatic? = #t,     // Default automatic shift
                           days = 3)            // Default 3-day rental
   ...
end method;
```

Notice the usage of **#t** and **#f**. These are the Dylan values for Boolean true and false, respectively.

Some examples of using this function are as follows:

```
rent-car(arnold, dallas, days: 7, sunroof?: #t);
rent-car(betty, dallas, days: 8, color: #"red");
rent-car(colin, vegas);          // Everything defaulted
```

You also have the option of specifying the return parameters for Dylan functions, as illustrated in Listing 4. This provides more information to the compiler to assist in optimization, as well as documents your code for other users. Dylan functions can return multiple values, which means the caller can get zero, one, or more than one value from the callee. This lets you program in a cleaner, more functional style than in C. In Dylan, you don't need to mix your input and output parameters and bash inputs to make them outputs, or clutter your code with definitions for funny data structures that do nothing more than carry the results of one function back to another.

Listing 4. Example of return declarations and multiple return values

```
define class <brick> (<object>)
   slot vert;
   slot horiz;
   slot depth;
   slot density;
end class;

define method calculate-weight (b :: <brick>)
   => weight :: <number>;                // Declares return parameter
   let (x, y, z) = bounding-box(b);      // Binds multiple values
   x * y * z * b.density;                // Returns one value
end method;

define method bounding-box (b :: <brick>)
   => (height :: <number>, width :: <number>, length :: <number>);
   values(b.vert, b.horiz, b.depth);     // Returns three values
end method;
```

All methods in a generic function must be *congruent*. Basically, this means they must all take the same number of required parameters, and they must agree on taking keyword and rest values. There are a few more options you can specify for a generic function using the **define generic** statement, which can also constrain method congruency.

MULTIPLE POLYMORPHISM

One interesting feature of Dylan is that functions are *multiply polymorphic* (unlike in C++ or Smalltalk). A function can have as many required parameters as you like, and any or all of them can be specialized. When you call a generic function, a method is picked based on the specializations of all the required parameters, not just of the first one.

There are two methods in the **profits** generic function defined in Listing 5. The second of these methods is more specialized than the first one, because its script parameter (**<sci-fi>**) is more specific than that in the first (**<script>**). It just happens that the script parameter is in the second position. When selecting the method to handle a call like **profits(betty, tender)**, Dylan determines that the first method is the only one that's applicable, so that's the one that's used (see Figure 3). It turns out that both methods are applicable in a call like **profits(arnold, zarx)**. The second method is more specific, so that's what gets invoked.

Listing 5. Multiply polymorphic functions

```
define method profits (star :: <actor>, script :: <script>)
   (star.fans * $ticket-price)           // Money from ticket sales
   - (script.fx-budget + star.salary);   //  minus expenses
end method;

define method profits (star :: <actor>, script :: <sci-fi>)
   next-method() / 2;            // Sci-fi is out of fashion these days
end method;
```

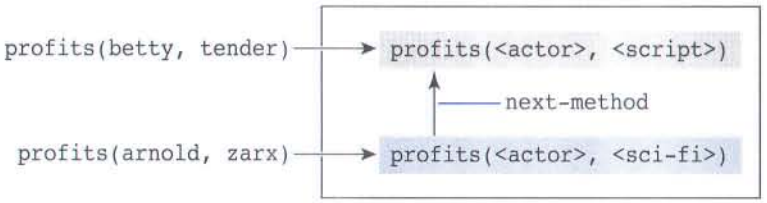


Figure 3. Method selection based on all arguments (not just the first one)

The body of the second **profits** method uses a special trick to inherit functionality provided in the base method. It calls **next-method**, a Dylan function that calls the next appropriate method in the generic function, in decreasing order of specificity. In the example, **next-method** gets a numeric value calculated by the first method, divides it by 2, and returns that to the caller. As a result, you don't have to write the basic equation twice; new methods have the option of calling up the specificity chain and doing what they want with the results. You can also add code to perform tasks before or after calling **next-method**.

THE ROLE OF MODULES

Dylan provides an important abstraction tool, called a *module*, which typically contains several related functions and classes. Modules let you simplify or limit access to objects by controlling their names. In other words, a module is a *namespace*, a set of names and the objects they refer to.

A module's definition specifies which names are exported. This gives you control over which variables, functions, classes, and slots are private to that module and which are public. For example, suppose the code in Listing 2 lived in a module called the **studio** module. We could define this module with the statement below, which exports three classes and three functions. Since **arnold** and **betty** are not exported, they're private to the **studio** module, and are inaccessible to any code outside it.

```
define module studio
   use dylan;
   export <project>, <actor>, <script>, name, name-setter, profits;
end module
```

Modules can selectively import some or all of another module's exports. Once imported, these can be used internally, extended, or reexported. We can define a new **hollywood** module that uses (imports everything exported from) the **studio** module. Notice that both modules also use the **dylan** module. Since the **dylan** module defines all the basic language primitives (like addition), it's a good idea for user-defined modules to always use it.

```
define module hollywood
   use dylan;
   use studio,
      export: name, profits;
   export <movie>, <tv-show>, <videogame>, do-oscars;
end module
```

This definition assumes that the **hollywood** module defines three new classes, plus one new function for computing the Oscar winners. It may define others for internal purposes, but those are the only internal classes and functions that it exports. The module also exports two functions imported from the **studio** module, **name** and **profits**. Even though the **hollywood** module imports the **<actor>** class from the **studio** module, there's no way to access the **salary** slot because **salary** wasn't exported, and hence cannot be imported into the **hollywood** module (see Figure 4).

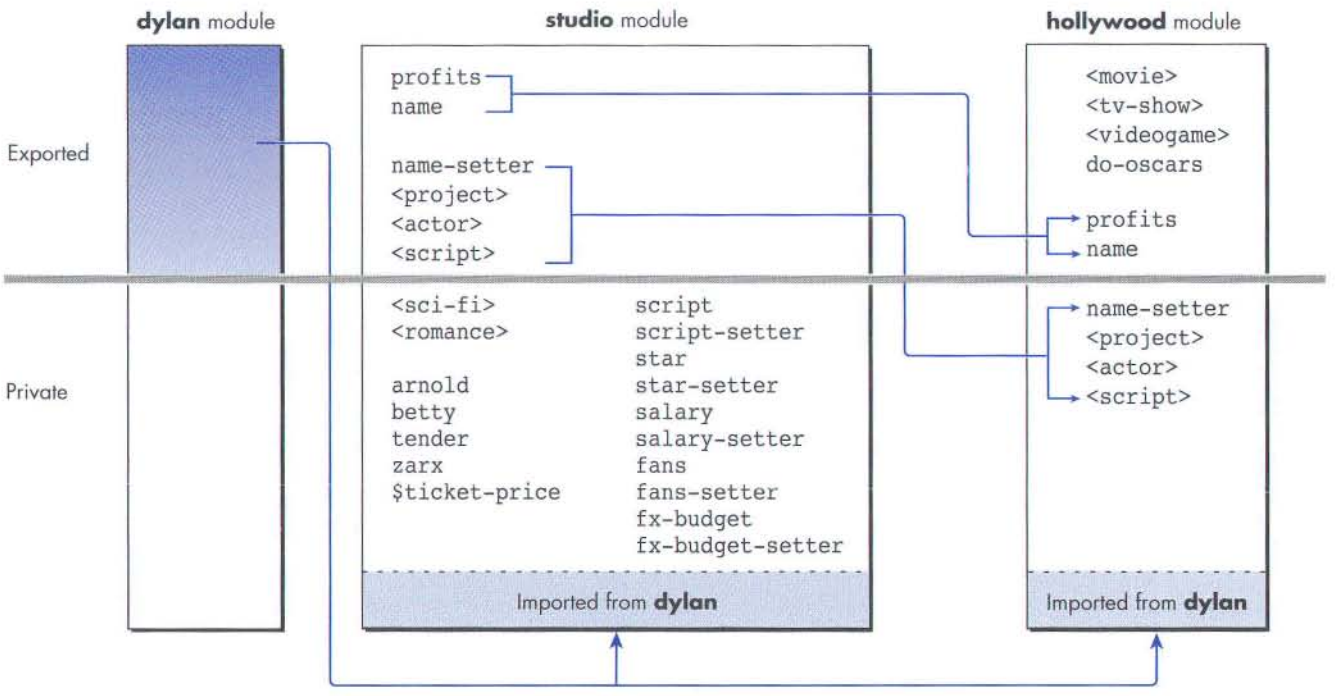


Figure 4. Selectively exporting names from modules to other modules

You can selectively export just the getter but not the setter function for a slot, which has the effect of making the slot read-only to all other modules. This is what **hollywood** does with **name**. Code in the **hollywood** module can change an object's

name because **name-setter** is imported from **studio**, but clients outside **hollywood** can only read, but not set, an object's name.

You could go ahead and define a new function in the **hollywood** module called **fans**, but it would have nothing to do with the **fans** slot in **<actor>**. This new **fans** function would be totally unrelated, and could have a different number of parameters than the **fans** function in the **studio** module. It's like two different cities each having a street called Main Street; the references are not valid across city borders. This is another key advantage of namespaces — they reduce the pressure to keep names unique at the expense of legibility or clarity.

You can even rename what you import, which is useful to prevent name conflicts, or to emphasize the origin of a name. For example, the following version of the **hollywood** module imports the **<project>** class from **studio**, but renames it. Within this **hollywood** module, the class is known only as **<production>**, not **<project>**. Modules have many more fancy renaming and import/export features, but we'll skip them for now.

```
define module hollywood
   use dylan;
   use studio,
      export: name, profits,
      rename: {<project> => <production>};
   export <movie>, <tv-show>, <videogame>, do-oscars;
end module
```

Modules let you control the interface to a portion of code by specifying exactly what you want to make public. You can even use several modules to provide high- and low-level interfaces to the same internal code — a capability not available in C++. For example, a **hollywood-tourist** module would import, rename, and export a subset of documented high-level calls to one set of users, whereas a separate **hollywood-insider** module might import, rename, and export more detailed calls to a different audience. This helps keep the implementation and interface nicely separated.

C++ has many notions of scope, including lexical (block scope inside functions), class, file, and name space. Some people even rely on the selective inclusion of header files or verbose name prefixes ("typographic scoping") to prevent name collisions. Dylan's simpler scheme — just lexical scope and modules — provides precise control over the importing, exporting, and naming of classes, functions, and variables in a clean and consistent way.

COMING SOON TO A DESKTOP NEAR YOU

In this whirlwind tour, you've had a quick look at how to write classes, functions, and modules in Dylan. Methods are grouped into generic functions, instead of being "owned" by classes. Modules package the names of related classes and functions into convenient APIs.

Apple Dylan isn't planned to ship until later this year, but that doesn't mean you can't play with Dylan before then. If you like what you've seen here, or want to see more, check out the goodies on the CD or those available from on-line services (see "Where to Get Dylan Software and Information").

Just like the Macintosh, Dylan was carefully designed from scratch to make your life a lot more fun and productive. Enjoy, and happy hacking!

WHERE TO GET DYLAN SOFTWARE AND INFORMATION

Some experimental freeware Dylan implementations are now available. Marlais, an interpreter, has been ported to Macintosh, Windows, and UNIX®, and is included on this issue's CD so that you can play with the code examples in this article. Mindy, a byte-code compiler, is available for UNIX. Also on the CD is the *Dylan Interim Reference Manual* and other goodies.

Other sources of Dylan software and documentation include the following on-line services:

- On the Internet, http://www.cambridge.apple.com is the Apple Dylan World Wide Web server, and ftp.cambridge.apple.com:/pub/dylan is Apple's Dylan ftp site.
- On AppleLink, look in Developer Support: Developer Services: Development Platforms: Dylan Related.
- On eWorld, go (Command-G) to "dev service"; then click Tool Chest: Development Platforms: Dylan Related.
- On CompuServe, type GO APPLE to get to the Apple support forum. There are 16 libraries; go into Programmers/Developers Library #15.

Dylan discussions can be found on the Internet newsgroup called comp.lang.dylan. You can also access Dylan discussions through e-mail. Internet users can ask to be included in discussions by sending a request to info-dylan-request@cambridge.apple.com (AppleLink users can use the address info-dylan-req@cambridge.apple.com@internet#).

If you'd like to become a beta tester of Apple Dylan, please send a message, including your name, address, telephone number, and a brief statement of what you'd like to do with Apple Dylan, to AppleLink DYLAN.

Thanks to our technical reviewers Stoney Ballard, Jeff Barbose, Ken Dickey, Phil Kania, Ken Karakotsios, David Moon, Carl Nelson, and Kent Sandvik.•

TIM MARONEY

MPW TIPS AND TRICKS

Launching MPW Faster Than a Speeding Turtle

One myth about MPW is that it's slow, but that's an unfair description. Personally, I think "glacial" would be a more appropriate word, or perhaps "executionally challenged." However, it's possible to speed it up in a variety of ways, such as simulating the 68020 instruction set on a fully loaded Cray. On a tighter budget, you can improve MPW's launch time just by making some minor changes to your configuration.

Perhaps some of you are asking, "Who cares about launch time? Compile speed is the important thing! God built the whole world in less time than it takes me to compile my project with PPCC, and he only had a slide rule!" It's a valid objection, and I'm glad you brought it up, but many people launch MPW more often than they compile. Quite a few projects use MPW as a development workhorse because of its scripting and source control capabilities, but compile and link using language systems that aren't laboring under the delusion that they're getting paid by the hour.

I didn't invent this technique, but I've tuned it up and eliminated some trouble spots. The original was distributed on a Developer CD so old that I can't find it now.•

STATES' RITES

The trick is simple and capitalizes on an important fact about MPW tools. Because of the innovative approach MPW takes to the traditional TTY interface, it's easy to execute the output of tools by selecting the output with the mouse and pressing the Enter key. Tool writers are strongly encouraged to write executable commands as their output. Since some of the tool writers didn't get the message, there are umpty-million exceptions, but when the tool does the right thing it's very useful.

There's an even better way to select the output, which is to press Command-Z twice after running the tool, but don't say I told you so. On the Macintosh, Undo followed by Redo is supposed to return you to your original state.•

The nice people responsible for the Set, Export, Alias, AddMenu, SetKey, CheckOutDir, and MountProject commands followed MPW policy and made them reversible: giving these commands without parameters dumps a list of commands that you can execute later to return to your current state.

As it happens, in a standard MPW configuration there's not much to your state beyond the output of these seven commands. You're in a current directory and some file windows might be open, and that's about all that matters. You can save the directory and the open files with four lines of script.

You can probably see where all this is leading. MPW lets you install scripts that get run when it quits and when it starts up. Is it really faster to save your state when you quit and restore it on your next launch than it is to iterate over your startup files? The answer is an emphatic yes, at least with the usual baroque MPW configuration. You'll see much less improvement if you're already using a lightweight MPW without many startup files.

Now if you're clever, you've probably written all kinds of things that need to get loaded each time you start up. I can understand that — I often feel like I need to get loaded every time I launch the MPW Shell myself! Maybe you've written a tool that lets you add hierarchical menus to the MPW Shell so that you can keep your wrist muscles toned, or a floating utility window with buttons for your frequently used commands. These clever hacks are going to hurt your startup time, but if you *must* do something every time you start up the Shell, you can move these commands into separate files that still get executed on each launch.

THAT DOES IT – I QUIT!

Saving the state is the easiest part of the trick. Just put a file named Quit in your MPW folder. You can overwrite the default Quit script if you have one, but if you need to keep it, you can name this script

TIM MARONEY is a leather-wearing vegetarian from Berkeley, California. He's been published in *MacTutor* and *Gnosis* magazines and in the *San Francisco Chronicle*. Tim is currently working as a contractor at Apple on the next release of the Macintosh operating system. When he's not standing on his head, he's usually peering at eldritch tomes such as the *R'lyeh Text* and the *SOM User's Guide*. No, that's not what we expected him to look like either.•

Quit•SaveState instead; the default Quit script will run it, as well as any other scripts named Quit•Whatever. The script should read like this, more or less:

```
# Quit and save state for fast startup

# We need to set Exit to 0 so that errors won't
# cause Quit or Startup to bomb, but we also want
# to maintain the user's setting of the Exit
# variable. Save and restore it.
Set SaveExit {Exit}
Export SaveExit
Set Exit 0

# State saving is turned off by creating a file
# named DontSaveState in the MPW folder.
If "`Exists "{ShellDirectory}"DontSaveState`"
   Delete -i "{ShellDirectory}"DontSaveState ∂
      "{ShellDirectory}"MPW.SuspendState ∂
      ≥ Dev:Null

Else

   # Write the state to a temporary file.
   Begin

      # Tell the restoration not to bomb.
      Echo Set Exit 0

      # Save the custom menus.
      AddMenu

      # Save the current directory.
      Echo Directory "`Directory`"

      # Save the open windows.
      Echo For window In "`Windows`"
      Echo 'Open "{window}" || Set Status 0'
      Echo 'End ≥ Dev:Null'

      # Save the aliases.
      Alias

      # Save the variables.
      Set

      # Save the exports. This runs much faster
      # with all the exports on one line, so we
      # use -s to get all the names at once.
      Echo Export "`Export -s`"

      # Save the key assignments.
      SetKey

      # Save lines that will execute the UserMount
      # script if any. The script doesn't have to
      # exist, and it's harmless to throw it away
      # between saving and restoring state.
      If "`Exists "{ShellDirectory}"UserMount`"
         Echo Execute ∂
            "{ShellDirectory}"UserMount ∂
            "≥ Dev:Null"
      End

      # Save the mounted Projector databases and
      # their checkout directories.
      MountProject
      CheckOutDir -r

      # After the rest of the state is restored
      # with Exit set to 0 to prevent bombing,
      # save lines to restore the user's setting
      # of Exit.
      Echo Set Exit '{SaveExit}'

   End > "{ShellDirectory}"MPW.SuspendState ∂
      ≥ Dev:Null

End

# Sometimes anomalies prevent the Worksheet from
# auto-saving at Quit time; make sure it does.
Save "{Worksheet}"
```

Every time you quit the MPW Shell normally, this Quit script will save your complete state to a file named MPW.SuspendState in your MPW folder. You probably noticed that this can be turned off by creating a file named DontSaveState. You don't have to do this by hand; if you'll just wait a gosh-darn minute, I'll give you a menu command for it.

Unfortunately, the Choose command, which lets you mount a file server, isn't reversible; that is, it doesn't put out a list of Choose commands that you could run later to remount your servers. Using this Quit script, though, you can create a file named UserMount in your MPW folder that will be executed every time you launch, before any attempt is made to remount your saved projects. This file should contain Choose commands that mount the servers on which your Projector databases are located. If you're not using Projector or other remote services, there's no reason to create this file. Here's an example, assuming I have a Projector database on the volume "Rendezvous" on the server "Development" in the zone "Engineering Heck":

```
if !"`Exists Rendezvous:`"
   choose -u 'Tim Maroney' -askpw ∂
      "Engineering Heck:Development:Rendezvous"
end
```

The Quit script isn't especially tricky, but if you're new to MPW scripting, you may be interested to note a few features.

First, observe the use of the back quote (`), that otherwise useless key at the top left of your keyboard. MPW uses it the same way as **csh** (pronounced "sea-shell"), the seminal UNIX shell from Berkeley: a command inside back quotes is executed and its output is inserted into the command line containing it. In this case, the Directory, Windows, and Export commands are backquoted, capturing their output so that it can be combined with other text using the Echo command. The Exists command is backquoted so that its output can be treated as a conditional expression.

Another handy fact is that compound statements, like Begin...End blocks, conditionals, and loops, are treated as single commands that can be redirected in their entirety. This saves a lot of needless repetition: you don't have to redirect each statement inside the block. Note the use of the error redirection operator "≥", typed as Option-period. Like UNIX, MPW Shell has separate output and error channels that can be redirected independently. In this Quit script, errors are redirected to yet another UNIXism, the faux file "Dev:Null," which is another way of saying send them to oblivion.

You can find out more about the various redirection options in MPW by starting up the MPW Shell and giving the command Help Characters. For clarity, the help text refers to the error channel as the "diagnostic file specification."•

One very important feature of MPW is its set of built-in variables. You can set up any variables you want by using the Set command, and expand them by putting them in curly brackets; there are also quite a few built-in variables that tell you things about the state of the MPW Shell and let you modify its behavior. The ShellDirectory variable is used extensively in the script; when expanded (**"{ShellDirectory}"**) it yields the path name of the folder containing the MPW Shell, where many useful things are stored. The old name for this variable is "MPW," which you can still use as a synonym.

Another built-in variable is Exit. If Exit isn't 0, script commands that fail will bring the execution of their script to a screeching halt; if it is 0, subsequent script commands will go on regardless of earlier failures, much like some people's conversational gambits at trade-show parties. These fast-launch scripts set Exit to 0, because if there's a failure at some point, the rest of the state should still be saved and restored. In normal MPW setups, Exit is set to 1 most of the time, but since idiosyncratic MPW configurations set it to 0 as the default, some special work is needed to save and restore the user's Exit setting. This is done by saving Exit in a custom variable named SaveExit, which records Exit at the beginning of the Quit script and restores it at the conclusion of the MPW.SuspendState script.

HAPPINESS IS A WARM BOOT

The startup sequence is slightly more complicated. After all, you've got to iterate over all those startup files sometime. The approach I'm using distinguishes between a cold boot, which does a pretty normal startup, and a warm boot, which starts up quickly from MPW.SuspendState.

"Cold boot" and "warm boot" are terms that old-time programmers will remember from the manual kick-starters on the original Model T computers.•

There's a menu item you can use to force the next launch to be a cold boot, or you can throw away the MPW.SuspendState file before launching for the same effect. The cold boot mechanism exists mostly for the sake of paranoia, so programmers tend to use it frequently. Generally speaking, you don't need to do a cold boot after you change your startup files; you can just select the change and press Enter. The modifications will get stored in the saved state the next time you quit.

MPW comes with a file named Startup that gets executed each time the Shell is launched. Rename Startup to ColdStartup and put the following in a new Startup file:

```
# Restore the state if possible; else cold boot.
# ∑∑ means redirect to end of file.
If "`Exists "{ShellDirectory}MPW.SuspendState"`"
   Execute "{ShellDirectory}MPW.SuspendState" ∂
      ∑∑ "{Worksheet}"
   Set ColdBoot 0
Else
   Beep 2g,3 2f,3 2a,3   # Hum a merry tune
   Begin
      Echo "MPW.SuspendState was not found."
      Echo "Here's your Cold Boot…"
   End ∑∑ "{Worksheet}"
   Execute "{ShellDirectory}ColdStartup"
   Set ColdBoot 1
End

Export ColdBoot

# Do anything that needs doing each launch
# (UserStartup·X files in EachBoot folder).
```

```
If "`Exists -d "{ShellDirectory}EachBoot"`"
   For fileName in ∂
      `(Files ∂
      "{ShellDirectory}"EachBoot:UserStartup·≈ ∂
         || Set Status 0) ≥ Dev:Null`
      Execute "{fileName}"
   End
   Unset fileName
End
Unset ColdBoot
```

The default Startup script runs all the files whose names start with "UserStartup•" in the MPW folder: UserStartup•Utilities, UserStartup•EraseBootBlocks, UserStartup•AlterPersonnelRecords, and so forth. You just moved the default Startup script to ColdStartup, so these files will get reexecuted whenever you do a cold boot. Also, in case you need to do something every time you launch regardless of whether it's a cold or a warm boot, you can put it in a UserStartup•Whatever file in a folder named EachBoot in the MPW folder.

Sometimes you need to do something different at startup depending on whether it's a cold or a warm boot. The Startup script above sets a variable named ColdBoot so that you can distinguish between cold and warm startups. In one of your startup scripts, you can use the ColdBoot variable in a conditional construct. For instance, suppose you're part of a large project with a centrally maintained MPW configuration that uses a custom tool named HierMenu to create a hierarchical menu. HierMenu is called from the central UserStartup•Project script at cold boot, but because it's not a standard part of MPW, it also needs to get called from an EachBoot script at warm boot — the state isn't automatically saved by the Quit script. You don't want to edit the shared file UserStartup•Project because you'll have to laboriously reapply your change every time the build engineers improve the central copy, but you can't run HierMenu more than once without bringing the system to its knees. The solution is to create a UserStartup•DoHierMenu file in your EachBoot folder which only runs HierMenu in the case of a warm boot, like so:

```
If ¬ "{ColdBoot}"
   HierMenu HierItem MainMenu 'Title for Item'
End
```

I promised you a menu command to do a cold boot. Here it is (in the immortal words of Heidi Fleiss, don't say I never gave you anything). Put this in a file named UserStartup•ColdBootItem in your MPW folder:

```
AddMenu File '(-' '' # menu separator
AddMenu File "Quit with Cold Boot…" ∂
   'confirm "Quit with cold boot?" && ∂
   (Set Exit 0; ∂
   Echo > "{ShellDirectory}"DontSaveState; ∂
   Quit)'
```

MEASURING PERFORMANCE WISELY

If you measure performance by elapsed time, MPW can be slow. However, real-world performance has more to do with usefulness than with theoretical throughput. I don't use my computer to run Dhrystone benchmarks: I use it to accomplish tasks. MPW gives me the power to accomplish the complex and bizarre tasks of programming automatically.

Real-world friendliness is always relative to a particular set of users and a particular set of tasks. The very things that make UNIX and MPW unfriendly to novice users make them friendly to programmers, who have the unusual skill of memorizing arcane commands and connecting them in useful ways. Don't get me wrong; MPW is not the final frontier of development environments. A true next-generation software authoring system would make command shells and project files seem equally ridiculous, but command-line interfaces for programmers are a sound approach, at least for now. And with a little tuning, they can be greatly improved. In future columns I'll be sharing more tips on making the worksheet a pleasant place to live.

Thanks to Dave Evans, Greg Robbins, and Jeroen Schalk for reviewing this column. Thanks also to beta testers Arnaud Gourdol, Jon Kalb, and Ron Reynolds.•

Designing a Scripting Implementation

Now that AppleScript is fast becoming an important core technology of the Macintosh operating system, more and more developers are making their applications scriptable or improving their scriptability. The way you design your scripting implementation can make the difference between satisfaction and frustration for users who want to script your application. The tips presented in this article will help you do it right.

CAL SIMONE

A well-designed user interface enables users to discover your application's capabilities and take full advantage of them. Likewise, the way you design your scripting implementation determines the degree of success users will have in controlling your application through scripting — writing simple, understandable, and, in most cases, grammatically correct sentences.

And just as the consistency of its user interface has been perhaps the most important factor in the Macintosh computer's ongoing adoption and success, consistency is an essential part of the world of scripting. It's highly important for users (by which I mean anyone who writes scripts, including power users, solutions providers, consultants, in-house developers, resellers, and programmers) to feel as if they're using a single language, regardless of which application they're scripting. As a developer, you have a responsibility to extend the AppleScript language in a consistent manner.

My purpose in this article, which might be considered a first attempt at some "human scriptability guidelines," is to offer conventions, suggestions, and general guidelines that you can follow to maintain consistency with the AppleScript language. I also give some suggestions for redoing a poorly done scripting implementation. (I'm assuming you're already convinced that you should make your application scriptable; if you're not, see "Why Implement Scriptability?") The result of doing all this work is that the AppleScript language feels consistent across applications of different types produced by different vendors.

CAL "MR. APPLESCRIPT" SIMONE (AppleLink MAIN.EVENT) has dedicated his life to bringing scripting to the masses. He can usually be found moving fast through the Worldwide Developers Conference or MACWORLD Expo, a cloud of dust in his wake. A founder of Main Event Software of Washington, DC, he designed the Scripter authoring and development environment for AppleScript and sometimes teaches AppleScript at corporate sites. An honorary member of the Terminology Police as a result of having reviewed scripting vocabularies for more than two dozen third-party products, Cal is available to look at *yours*. He lives about a mile from the White House and was fond of saying of President Bush, "I don't bother him, and he doesn't bother me." •

WHY IMPLEMENT SCRIPTABILITY?

If you're still wondering why you should implement scriptability in your application, consider these reasons:

- Scripting gives users a way to control your application through a different interface. This alternate interface allows users to incorporate your application into multi-application scenarios, as well as to automate tedious, repetitive tasks.
- Allowing your application to be controlled through Apple events enables Apple Guide to give your users truly active assistance.
- Implementing scripting prepares your application for OpenDoc by ensuring that your part handlers will be able to mesh smoothly with other parts.
- Making your application scriptable ensures that as speech recognition matures, you'll be able to give users the option of voice control.

It's important to implement AppleScript support in your core application, rather than through an external API, as some databases such as 4th Dimension and Omnis do. When your core application isn't Apple event–aware, two things happen: (1) no dictionary resides in the application itself, and (2) functionality is usually limited. Users have difficulty doing decent scripting of these applications, by and large. If you simply *must* support Apple events through an external API, at least support the dynamic terminology mechanism for your extensions.

The bottom line is this: If your application isn't scriptable soon, you'll be left out in the cold. If you do the work now, not only will you open up more uses for your application in the "big picture," but you'll also be that much closer to implementing what you need in order to support several other technologies. So please, don't put it off!

FIRST, SOME BASIC CONCEPTS

A good scripting implementation consists of two parts:

- An Apple event *object model hierarchy*, which describes the objects in your application and the attributes of those objects.
- A *semantic vocabulary*, also called a *terminology*, consisting of the terms used in the construction of command statements. Your vocabulary is stored in your application's 'aete' resource, known to users as the *dictionary*.

Your terms, and the organization of those terms in your dictionary, directly affect the ability of users to explore and control your application through scripting. Creating a vocabulary through which users can effectively and easily script your application takes time and careful effort. Don't expect to spend six months implementing Apple events and then simply to throw together a dictionary at the last second.

It's important to note that a well-designed Apple event structure greatly increases the ease of scripting your application. In a minute I'll say more about that, but first let's look at the basic anatomy of an AppleScript command.

ANATOMY OF A COMMAND

You should design your scripting implementation so that users will be guided into using a clean, natural-language sentence structure. To help you begin to visualize the kinds of sentences your users should be encouraged to write, let's look at AppleScript's syntactic statement structure (say that three times fast!). All application-defined commands are in the form of imperative sentences and are constructed as follows:

verb [noun] [keyword and value] [keyword and value] . . .

These elements of sentence construction can be thought of as parts of speech that make up a human-oriented computer language. Here are a couple of examples of commands:

```
close the front window saving in file "Goofballs:Razor"
set the font of the first word in the front window to "Helvetica"
```

Let's dissect these:

`close`	verb, corresponding to kAECloseElement
`the front window`	noun, corresponding to keyDirectObject (typeObjectSpecifier)
`saving in`	keyword, corresponding to keyAEFile
`file "Goofballs:Razor"`	value, of typeFSS
`set`	verb, corresponding to kAESetData
`the font of the first word in the front window`	noun, corresponding to keyDirectObject (typeObjectSpecifier)
`to`	keyword, corresponding to keyAEData
`"Helvetica"`	value, of typeWildCard

Note that for application-defined commands, a *verb* — for example, **close** or **set** — is the human language representation for the action described by an Apple event (which I often shorten to just *event*), so there's a general correspondence between Apple events and verbs. In this article, I identify Apple events by the event's name, its 4-byte ID, or the constant name for the ID. For example, the Close Element event has the ID 'clos' and the constant name kAECloseElement, and corresponds to the AppleScript verb **close**; the Set Data event has the ID 'setd' and the constant name kAESetData, and corresponds to the AppleScript verb **set**.

Your ability to guide users toward writing clean, natural-language statements depends a great deal on your use of the object model, as I explain next.

WHY USE THE OBJECT MODEL?

Supporting the object model facilitates scripting by allowing the use of familiar terms for objects and actions. In the last couple of years, some important applications that don't implement the object model have shipped, and most of them range from difficult to impossible to script. Let's explore a couple of examples of how using the object model can make scripting a lot easier.

Apple events and the object model are covered extensively in "Apple Event Objects and You" in *develop* Issue 10 and "Better Apple Event Coding Through Objects" in Issue 12.•

The following script is the result of a lack of defined objects in the application we'll call My Charter. The lack of defined objects leads to a vocabulary in which every noun-verb combination must be covered by verbs alone — a vocabulary that doesn't relate to other applications and that forces users to learn a new set of commands.

```
tell application "My Charter"
   Plot Options myOptions
   Set Axis Lengths for X 100 for Y 100
   Output PICT
   Plot chart "pie"
end tell
```

By contrast, the script below describing the same operation in much more familiar terms results when the application uses familiar objects and characteristics of objects (properties):

```
tell application "My Charter"
   make new chart
   tell chart 1
      set the type to pie
      set the x axis to 100
      set the y axis to 100
   end tell
end tell
```

As illustrated by this script, a principal indication of solid use of the object model is that the most common verbs used in scripts are **make**, **set**, and **get**.

Users are more likely to remember the terms for objects than commands. Moreover, from the user interface, they often use Command-key shortcuts for the actions instead of looking at the menu items once they get comfortable using your application. If you don't implement the standard commands, they'll probably need to go back to your application's menus to find out that the menu command is, for instance, Plot Chart. You can help them by making the scripting terms intuitive. For instance, they already know what a chart is, and they're familiar with the standard AppleScript verbs **make** and **set**, which they're using to script other applications. Thus, the second script above will feel like an extension of the same language used in scripting other applications, while the first script won't.

Now consider this partial list of custom verbs from a popular mail application that doesn't follow the object model:

AddAttachment	SetSubject	GetSubject
AddTo	SetText	GetText
AddCC	SetReceipt	GetReceipt
AddBCC	SetPriority	GetPriority
AddToAtPO	SetLog	GetLog

Notice some patterns here? All of them start with Add, Set, or Get — and this isn't even a complete list of all the commands in this application starting with these verbs. It's definitely time for this application to go with the object model. Most of the above commands can be replaced by **set** and **get** commands applied to properties such as **subject**, **receipt**, **priority**, **log**, and so forth.

DESIGNING YOUR OBJECT MODEL

Now that you know how important the object model is to scriptability, let's look at how to get started with your design. As you approach the design of your object model, keep in mind both your application's objects and the style of the commands you expect your users to write.

DECIDE WHICH OBJECTS TO INCLUDE

Base the design of your object model only partly on your application's objects. Keep in mind that the objects in an object model aren't necessarily the same as the programmatic objects in an object-oriented program but rather represent tangible objects that the *user* thinks about when working with your application.

Generally, you won't want the user to script interface elements, such as dialog box items (whose meaning should be expressed through verbs, or properties of the application or your objects), but rather objects that either contain or represent the user's data (which I'll call *containers* and *content objects*). For example, an object model might incorporate documents (containers); graphic objects (containers or content

objects); forms (containers) and the elements of a form, such as fields (content objects); cells in a spreadsheet or database (content objects); and text elements, like paragraphs, words, and characters (content objects).

You should think carefully about whether to make something an object or a property; this is discussed later in the section "Other Tips and Tricks."

THINK FROM ACTIONS TO OBJECTS

When you design your commands, the primary thing to keep in mind is how you want the script command statements to read or to be written. The style of the commands you expect your users to write should determine your object model, *not* the other way around.

As programmers, we have the notion that an object "owns" its methods; we think in terms of sending messages to an object. For instance, the following C++ code fragment sends several messages to one object:

```
CDocument::Print
CDocument::Close
CDocument::Save
CDocument::Delete
```

By contrast, users think about doing some action to an object. So when you design your commands, you should think about allowing verbs to be applied to many different types of objects, as illustrated here:

```
print document "Fred"
print form ID 555
print page 4
```

Examine the actions that users take with your application and the objects that the actions are taken on. This will lead you naturally to an effective object model design.

START — BUT DON'T END — WITH MENU COMMANDS

One place to start your scripting implementation is to implement your menu commands as verbs for scripting. You can use this as a push-off point, but because your menu commands most likely don't supply all the functionality of your application, you shouldn't limit yourself to *only* implementing menu commands.

Before I say any more about this approach, you should note these two very important caveats:

- Keep in mind that the philosophy of AppleScript is to allow the user to script the *meaning* behind an action, not the physical act of selecting a menu item or pushing a button. This perspective should be the foundation for your entire design.
- When you use the standard events, often there's a **set <property>** scripting equivalent that's better than creating a new verb to match a particular menu item. Menu commands are designed for user interface work and don't always provide the best terminology for scripting. Thinking in terms of **make**, **set**, and **get** can often be more useful than creating verbs that mimic menu commands.

That said, let me elaborate on the idea of implementing menu commands and beyond.

Ideally, you should allow users to achieve through scripting everything that they can with your user interface. To accomplish this, you should think of capabilities you would like users to be able to script that go beyond your menu commands, such as capabilities accessible only from tools in a palette or actions resulting from a drag and drop operation. On the other hand, it's not entirely necessary to make the capabilities available from your user interface identical to those controllable through scripting. Scripting is a different interface into your program, so it's OK to do things a bit differently.

For example, you don't have to create exactly one script statement corresponding to each user action. If a single menu item or button in your application results in a complex action or more than one action, it might produce clearer scripting or give more flexibility to allow the user to perform individual portions of the action through separate statements in a script. Conversely, it can also be better to combine more than one action into one statement, especially when the set of actions is always performed in the same sequence.

Also, actions that aren't even possible from the user interface can often be made scriptable. For example, the Scriptable Text Editor allows a script to make a new window behind the front window, something that the user normally can't do. You could also provide a method of accomplishing a task that's too complex or impossible to express through manipulation of objects on the screen.

MAKE AN EARLY BLUEPRINT

These two exercises can help you get started with designing your hierarchy and your command scheme:

- Write down in real human sentences as many commands as you can think of to control your application. Refer to these sentences later when you're thinking about what Apple events and objects to include in your implementation.
- Make an early version of your 'aete' resource (see "Tools for Developing an 'aete'"). You can then do your coding based on this resource.

TOOLS FOR DEVELOPING AN 'AETE'

To assemble your 'aete', you can choose from these tools:

- The aete editor stack — This HyperCard stack is a commonly used tool. It's a good way to assemble your 'aete' if it's not too large.
- The Rez files — Rez source files can easily be changed and can handle any size 'aete', so this is the tool of choice for developers who do serious work with resources. You'll need AEUserTermTypes.r and AERegistry.r as include files. In addition, you can refer to AppleEvents.r, AEObjects.r, AEWideUserTermTypes.r, and ASRegistry.r. You can use EnglishTerminology.r and EnglishMiscellaneous.r to examine the standard registry suites.
- Resource editors — Any resource editor *except* ResEdit will suffice. This is one situation in which ResEdit isn't really useful unless your 'aete' is microscopic; you can't open your resource using the 'aete' template if it's more than about 2K in size. Resorcerer includes a pretty decent 'aete' editor, considering the complexity of this resource — but be warned, the editor is equally complex.

The aete editor stack and the include files for Rez are available on this issue's CD and as part of the AppleScript Software Development Toolkit from APDA. Resource editors with good 'aete' editors are commercially available.

Details of the structure and format of an 'aete' resource can be found in Chapter 8 of *Inside Macintosh: Interapplication Communication*.

I would recommend that you go back and do both of these exercises again periodically throughout your development cycle. Use the combination of your 'aete' resource and the sentences as a blueprint during your implementation work.

MAKE THE CONTAINMENT HIERARCHY OBVIOUS

Your object model design includes an *object containment hierarchy*, a scheme indicating which objects are contained in which other objects. When you design your containment hierarchy, think again about the user's experience when writing scripts. Make it easy for the user to determine that objects of class *y* are contained in objects of class *x*, which is in turn contained in the application.

For instance, Figure 1 shows part of the object containment hierarchy for an imaginary application that contains text windows, folders, and a connection. The windows can contain one or more paragraphs, words, or characters; paragraphs can contain words or characters; and words can contain characters. Note that even though only one connection is possible for this particular application, **connection** is an object class contained by the application, as opposed to being merely a property of the application.

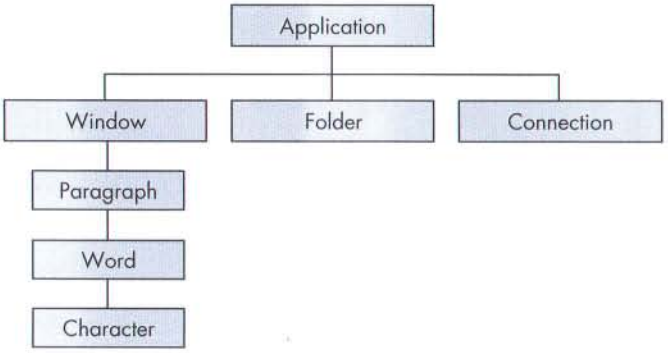


Figure 1. Part of a typical object containment hierarchy

It's important to connect up all the appropriate pieces of your containment hierarchy. It's *especially* important to hook up the main classes of objects — such as windows, documents, and other special objects not contained by other objects — to the *top level* of the hierarchy by listing them as elements of your application. Never "orphan" a class! Every object class (except the application) must be listed as an element of *something*. Most classes or objects are contained by another object. If any object can't be contained by another object, it *must* be contained by the application.

ASSEMBLING YOUR VOCABULARY

After you've taken a shot at writing down the kinds of commands suggested by your application's capabilities and the object model, it's time to think about how to assemble your vocabulary.

The AppleScript terms (commands, objects, and properties) that you'll use in your vocabulary fall into two categories:

- standard terms — those drawn from the standard Apple Event Registry suites and other well-defined suites
- extended terms — those you'll create to represent actions or objects specific to your application

To ensure that your scripting implementation will have as much consistency across applications as the user interface, you should use the standard terms whenever possible. As you've seen, this is inextricably tied to good object model design. See "Registry Suites" for descriptions of the standard suites. Unless you have a excellent reason, don't vary from the standard terms associated with these suites.

REGISTRY SUITES

The Apple event suites listed below (which include those defined in the *Apple Event Registry* as well as additional standard suites) are collections of events, objects, properties, and other terms common to most applications. For the sake of consistency with other scripting implementations, you should draw on these suites as much as possible as you design your vocabulary.

- The Required suite (kCoreEventClass = 'aevt') consists of the four events that the System 7 Finder uses to launch and terminate an application and to open and print documents. Note that while the Required suite's ID is 'reqd' (kAERequiredSuite), its four Apple events have the suite ID 'aevt'. Note also that in the early days, Apple originally referred to the Apple events in the Required suite as the core events (even including "core" in the C and Pascal constant names), creating some confusion with the Core suite. Please *don't* refer to the events in the Required suite as "core events."
- The Core suite (kAECoreSuite = 'core') consists of 17 events (14 main and 3 extra) and 8 objects that encompass much of the functionality that most applications support, including creating, deleting, opening, closing, and counting objects, as well as getting and setting properties. In an object model–based application, a great deal of the work in AppleScript is done through the Apple events in the Core suite. See the Scriptable Text Editor's dictionary for an example of the standard implementation of this suite. Applications generally support most but not all of the Core suite. Note that the Core suite's ID is 'core', and while most of its events have that suite ID, the Open, Print, and Quit events have the suite ID 'aevt'.
- The Text suite (kAETextSuite = 'TEXT') defines the object classes used in text handling, such as characters, words, and paragraphs, normally the direct objects of events defined in the Core suite. No Apple events are defined in this suite.
- The Table suite (kAETableSuite = 'tbls') defines the essential object classes used in table handling, such as rows, columns, and cells, normally the direct objects of events defined in the Core suite. Again, no Apple events are defined in this suite.
- The Database suite (kAEDBSuite = 'dbst') consists of the Group and Sort events; transaction-related events; the host, DBMS, database, session, and key objects; and extended definitions for the Table suite objects. It focuses the functionality of the Table suite specifically toward database activity.
- Miscellaneous Standards (kAEMiscStandards = 'misc') is a collection of additional Apple events, including editing events such as Cut, Paste, Undo, Redo, Select, and Revert, and the menu, menu item, and text item objects. This isn't used as a suite; only individual events or small groups of events are used.

Other Apple event suites that are used less frequently include the following:

- the Scheduling suite, used for applications such as calendars, appointment books, and alarm programs
- the Telephony suite, used by any application that handles phone numbers, including PIM, database, forms, and scheduling applications
- the Mail suite, based on the AOCE Mailer and used in mail-capable applications to mail documents
- the Collaborative Information suite, used in applications that access AOCE catalog services or manage contact or human resources information
- the System Object suite (not actually a suite), used for terminologies defined in Apple's scripting additions

The Word Services, QuickDraw, and QuickDraw Supplemental suites are generally *not* used in scripting.

To look up the accepted human-language constructs for the Required, Core, Text, Table, and QuickDraw suites, see the file EnglishTerminology.r (also available for French and Japanese); for the Database suite, see the file Database.aete.r; and for Miscellaneous Standards, see EnglishMiscellaneous.r (also available for French and Japanese). These files, which present the standard terms in the form of 'aete' resource templates (in Rez form), can be found on this issue's CD and are included in the AppleScript Software Development Toolkit.

USING STANDARD TERMS

When it comes to implementing the standard suites, you have three options:

- supporting an entire suite as is
- supporting an entire suite and overriding or adding to it
- supporting part of a suite

Supporting an entire suite. When you want to support *all* the events, parameters, classes, properties, and so on, of a suite, you should include the entire suite in your 'aete' resource. Listing 1 is an example of the Rez code you'll use to indicate that an entire suite (in this case, the Required suite) is supported. The four empty arrays in this listing are indicative of the fact that when you want a whole suite intact, you don't supply any events, classes, and so on. The entire suite will appear in your dictionary.

Listing 1. Sample Rez code supporting an entire suite

```
"Required Suite",    /* The entire suite, as is */
"Terms that every application should support",
kCoreEventClass,     /* 'reqd' */
1,
1,
{  /* array Events: 0 elements */
},
{  /* array Classes: 0 elements */
},
{  /* array ComparisonOps: 0 elements */
},
{  /* array Enumerations: 0 elements */
},
```

Note that whenever you use the 4-byte suite ID for a *suite itself* (as opposed to the suite ID for the individual events in a suite), *all* the standard definitions for that suite will automatically appear in your dictionary. Do *not* use this technique if you're implementing only a few of a suite's Apple events or objects. And note that this technique works only for the Required, Core, Text, Table, and QuickDraw suites, which are in AppleScript's 'aeut' resource. For all other suites, you'll need to include all the details of the suite in your 'aete' resource if you support it in its entirety.

Supporting only the Required suite doesn't qualify your application as Apple event–aware or scriptable. To qualify as being scriptable, your application must support more than just the Required suite.•

Supporting an entire suite to be modified. When you want to support a whole suite and then add to or otherwise modify it, use the Rez code in Listing 2 as a model. In this example, the entire Core suite is supported, and a new **copies** parameter is added to the **print** command. You can use the same technique to add property definitions to a standard object class. Just as in the previous example, here we don't specify any of the suite's details except the ones we're overriding or adding.

Supporting part of a suite.
On the other hand, when you want to implement only part of a suite, you need to explicitly define the subset of the suite's events and objects that you support. For

Listing 2. Sample Rez code supporting an entire suite to be modified

```
"Standard Suite",    /* The entire suite, plus an extra parameter */
"Common terms for most applications",
kAECoreSuite,        /* 'core' */
1,
1,
{  /* array Events: 1 element */
   /* [1] */
   "print",          /* This is the event being extended. */
   "Print the specified object(s)",
   kCoreEventClass,
   kAEPrint,
   ...
   {  /* array OtherParams: 1 element */
      /* [1] */
      "copies",      /* This is the parameter being added. */
      'NCOP',
      'shor',
      "The number of copies to print",
      ...
   }
},
{  /* array Classes: 0 elements */
},
{  /* array ComparisonOps: 0 elements */
},
{  /* array Enumerations: 0 elements */
},
```

example, let's say you implement only seven of the events in the Core suite (which nearly everyone implements only partially; these seven are the minimum you should support). You'll create a new suite with a unique ID — your application's signature, perhaps, or, as used by the Scriptable Text Editor, 'CoRe' (note the alteration from all lowercase, which prevents the whole Core suite from appearing automatically). Then you'll include the events and objects you want. Listing 3 shows how to do this in Rez code. Note that you should retain the original suite ID of 'core' for the individual Apple events (except for Open, Print, and Quit, which get 'aevt', as mentioned earlier in "Registry Suites"), both in your 'aete' and in your Apple event handlers.

The format for Rez listings in *Inside Macintosh* puts one element on each line, as I've done in Listings 1 and 2. To conserve space, I'll now begin putting more elements on each line, which is also a permissible format.•

USING EXTENDED TERMS

Whenever possible in your scripting implementation, you should use constructs and terms that are already in use. But sometimes you need to express concepts unique to your application. When you do, it's important to keep in mind the style of what's already been done in the AppleScript language, and in other applications.

The terms you create that aren't in the standard suites are actually extensions to AppleScript. The nature of these terms will directly affect the experience your users will have in scripting your application. You should create terms that give users the feeling that they're working within a unified language.

Listing 3. Sample Rez code supporting a partial suite

```
"Subset of the Standard Suite", /* Only seven of the Core events */
"Common terms used in this application",
'CoRe',     /* Note uppercase alteration of the 'core' suite ID. */
1,
1,
{  /* array Events: 7 elements */
   /* [1] */
   "count", "Return number of elements of a particular class ...",
   kAECoreSuite, kAECountElements, ...
   /* [2] */
   "delete", "Delete an element from an object",
   kAECoreSuite, kAEDelete, ...
   /* [3] */
   "exists", "Verify if an object exists",
   kAECoreSuite, kAEDoObjectsExist, ...
   /* [4] */
   "get", "Get the data for an object",
   kAECoreSuite, kAEGetData, ...
   /* [5] */
   "make", "Make a new element",
   kAECoreSuite, kAECreateElement, ...
   /* [6] */
   "quit", "Quit an application program",
   kCoreEventClass,  /* Open, Print, and Quit have 'aevt' suite ID. */
   kAEQuitApplication, ...
   /* [7] */
   "set", "Set an object's data",
   kAECoreSuite, kAESetData, ...
},
{  /* array Classes ...
},
...
```

Keep in mind that creating new object classes or properties is generally better than creating new verbs. If you do need to create your own verbs or use terms unique to your application, it's better to try to do it in the spirit of what's been done before instead of inventing your own "language within a language." Users shouldn't feel as if they're jumping between what appear to be separate "pseudo-languages" for each application.

Although early documentation from Apple suggested creating one custom suite containing your Core suite subset lumped together with your custom verbs, I don't always recommend this. If you're adding a lot of vocabulary, either new events or objects, you can make your dictionary more understandable by keeping the Core subset in one suite and defining your own new verbs in a separate suite. In fact, it's OK to make more than one custom suite if you have a great many new verbs or objects and if you can separate them into distinct functional groupings.

Make sure that the names for your new suites clearly indicate that they're custom suites or specific to your application. And when you create ID codes for your new events, objects, and such, remember that Apple reserves the use of all 4-byte codes

A WORD ABOUT DO SCRIPT AND DO MENU — DON'T!

One of the easiest methods of gaining the appearance of scriptability is to implement the Do Script event. Do Script enables users to pass statements or groups of statements written in your own internal scripting language to your application for execution. If you have an internal scripting language already, Do Script can be OK as a first step. Just don't stop there — in the end, it's useful as a supplement to the rest of your scriptability, but not as a substitute.

The drawbacks to Do Script are that (1) new users must learn a new language — yours — in addition to AppleScript, and (2) Do Script is a one-way communication in most cases — the script can control your application, but it acts much more like a puppeteer than a team leader. In the end, Do Script defeats the purpose of a single language for controlling all applications.

Another easy method of appearing to be scriptable is to implement a Do Menu event, in which a user can simulate pulling down a menu and selecting menu items. Again, this is no substitute for real scriptability.

By the way, if you're thinking about creating a *new* scripting language internal to your application, think again. The world doesn't need yet another private application-specific language. AppleScript is there for you, with all of its rich expressiveness, to use as your own. The benefit is that by the time you complete your scripting support, many of your users will already be familiar with AppleScript.

that contain only lowercase letters, so you should use at least one uppercase letter in the codes. There isn't yet a way to register your codes, but the Webster project (described at the end of this article in "Resources") aims to serve that end.

CONVENTIONS, TIPS, AND TRICKS

Here are some concepts and techniques that you can use to make your vocabulary more helpful to the script writer. Included are well-known tricks as well as techniques that aren't often considered. Adhering to these guidelines will make scripting cleaner and promote a consistent language "look and feel" across applications.

STYLISTIC CONVENTIONS

Begin terms with lowercase.

Begin all the terms in your dictionary with lowercase letters, except for proper names like PowerTalk. It may seem trivial, but it's actually quite important. If you use uppercase letters to begin your object names, for example, you'll end up with strange-looking commands that contain a mixture of uppercase and lowercase letters:

```
make new History
set the Title of the first History to ...
```

Using all lowercase letters gives a more consistent look:

```
make new history
set the title of the first history to ...
```

Separate all terms.

If you have terms that consist of more than one word, separate the words. Don't turn them into Pascal-like names:

```
ReplaceAll
set the TransferProtocol to ConvertFromMainframe
```

Instead, make them flow naturally:

```
replace all
set the transfer protocol to convert from mainframe
```

Use familiar terms, but avoid reserved words.
Generally speaking, you'll want to identify your object classes with terms your users are already familiar with. When it comes to your verbs, you can use many of your menu items, and for the rest use terms that will be familiar and that lend themselves to starting clean and natural statements. Plain human language is always preferable to C- or Pascal-style identifiers.

On the other hand, when you attempt to use familiar terms, keep in mind that the list of words that could potentially conflict with your dictionary is constantly growing and also depends on which scripting additions and applications are currently running on a particular computer. As a result, there's no official list of reserved words to avoid. Choose your terms with extreme care — remember, you're actually extending the language and what you do here will affect the future.

In summary, try to provide words that are familiar to users without running into conflicts with existing terminology. Don't make up new terms to express something when there's a clean way to do it using existing terminology: where possible, use terms analogous to those already in use to represent constructs (verbs, parameters, objects, properties, and enumerators) in your application. Conversely, don't use existing terms to represent something that differs from a term's accepted use.

ENUMERATIONS, LISTS, RECORDS, AND TYPE DEFINITIONS

Use lots of enumerations.
Very few developers have made effective use of enumerations. An *enumeration* is a set of constants, usually representing a fixed set of choices. In AppleScript, these constants, known as *enumerators*, are identified (like everything else) by 4-byte ID codes. Use an enumeration as the type for a parameter or property whenever there's a choice to be made from a specific list of possibilities, and make sure you use natural language.

For example,

```
set status to 1
```

or

```
set status to "warm"
```

isn't as helpful to the script writer as

```
set status to warm
```

This subtle change makes a great deal of difference. In the dictionary, the enumeration is displayed as "hot | warm | cool | cold," as opposed to "integer" or "string," and the user can easily see there's a choice. To accomplish this, you would create an enumeration with the enumerators **hot**, **warm**, **cool**, and **cold**, and use the 4-byte enumeration ID as the type for the **status** property of the class, as shown in Listing 4. The dictionary entry for this property will read "status hot | warm | cool | cold," instead of "status integer" or "status string."

It's an extremely common mistake among developers to try using ordinal values as enumerators, but it simply won't work. Unlike in C or Pascal, you can't use ordinal values — you must use 4-byte ID codes.

Listing 4. Creating and using an enumeration

```
{  /* array Properties: ...
   /* [1] */
   "status",
   'Psta',        /* Note uppercase in your IDs. */
   'Esta',        /* The enumeration's ID */
   "the status",
   reserved,
   singleItem,
   enumerated,    /* Use "enumerated" */
   ...
},
...
{  /* array Enumerations: 1 element */
   /* [1] */
   'Esta',
   {  /* array Enumerators: 4 elements */
      /* [1] */
      "hot", 'Khot', "A hot condition",
      /* [2] */
      "warm", 'Kwrm', "A warm condition",
      /* [3] */
      "cool", 'Kcoo', "A cool condition",
      /* [4] */
      "cold", 'Kfrz', "A cold condition"
   }
},
```

Set the list flag to indicate lists in parameters and properties.
If you're normally expecting a list of items as a parameter or a property, set the **list** flag (kAEUTListOfItems) in the parameter or property definition flags; the dictionary entry will then show "list of <whatever>." (Note that this is different from defining a parameter's or a property's type as **list**, which you should do when you want to indicate a mixed-type list or a list of lists.) An interesting possibility is to combine lists with enumerations, to indicate that the user can specify more than one choice, as in

```
set the applicability of filter 1 to {incoming, outgoing, ...}
```

Define record labels in a record definition.
To document the labels for the elements that make up a record, create a record definition in your dictionary. A record definition is actually a fake "class" in which the "properties" represent the labels in the record. Although there won't really be any objects in your application with this record type's class, your users can determine what labels are appropriate in order to fill in a record used as a parameter or a property value. Record definitions can also be helpful for users to interpret a record passed back as a result.

To create a record definition, invent a name for your record type and create a new class in your 'aete' resource with the record type name as the class name. Define all the possible labels as properties. As an example, Listing 5 shows the "class" definition you would create in your 'aete' resource for a record that looks like the following:

```
{name:"Fred", age:3, status:warm}
```

In this case, you would also define the enumeration for **status** with the enumerators **hot, warm, cool**, and **cold**. The record type would appear in the dictionary as follows:

```
class person info: A record containing information about a person
   person info
      name string -- the name
      age short integer -- age in years
      status hot|warm|cool|cold -- current status
```

Since a record definition is an "abstract class," it should be placed in the Type Definitions suite, described in the next section.

Listing 5. Class definition for our sample record definition

```
{  /* array Classes: 1 element */
   /* [1] */
   "person info", 'CPIN',
   "A record containing information about a person",
   {  /* array Properties: 3 elements */
      /* [1] */
      "name", 'pnam', 'itxt', "the name",
      reserved, singleItem, notEnumerated,
      ...
      /* [2] */
      "age", 'AGE ', 'shor', "age in years",
      reserved, singleItem, notEnumerated,
      ...
      /* [3] */
      "status", 'Psta', 'Esta', "current status",
      reserved, singleItem, enumerated,
      ...
   },
   {  /* array Elements: 0 elements */
   },
}
```

Put abstract class and primitive type definitions in special suites. There are two suites you can use to organize your dictionary better: the Type Definitions suite and the Type Names suite. These suites are used in special situations where you want to define object and type classes that are used in your terminology but that won't ever be actual instantiable objects in your application.

In the case of the record definition classes described in the previous section, you need to define abstract classes that don't refer to real objects. You'll also need to do this in the case of extra classes defined for property inheritance, which aren't instantiable as objects in your application. To include these record or type definitions, create a Type Definitions suite (also known as an Abstract Class suite) with the ID 'tpdf' (kASAbstractClassSuite; note that this constant isn't defined in any .r files, so you'll need to define it yourself) and include your abstract class and record definitions.

On some occasions you may want to add terms to your vocabulary that you don't want to show up in your dictionary. For example, you might need to provide the terms for primitive types, such as **integer** and **point**, to make AppleScript work properly, but users are already familiar with these elemental terms and don't need to see them defined. In this case, make a Type Names suite with the ID 'tpnm' (kASTypeNamesSuite) and include your types as classes in this suite. Well-behaved editors such as Apple's Script Editor and Scripter from Main Event will suppress the display of this suite.

To sum up, if you want these definitions to be visible to the user, include them in your Type Definitions suite. If you want them to be hidden, include them in the Type Names suite. Use of these suites will help keep the rest of your suites less cluttered.

NOTES ON DIRECT OBJECTS

Be explicit about direct objects.

Some developers have relied on a default or current target, such that commands that don't include a specific object target will act on the frontmost window or the last explicitly set object. There are three reasons to be careful here:

- Users of multiple applications may be confused by different assumptions surrounding the notion of a current object used as the target.
- If your Apple events act just on the current object, your users can only act on some other object by explicitly making it the current object. In the case where the current object is considered to be the frontmost window, there's no way to script other windows.
- Another script (or the user!) could make a different object the current object while a script is running.

The moral of this story is that it's best to be explicit at all times about the object that will be acted on.

Make the target the direct object.

One of our goals in scripting is to maintain a natural imperative command style throughout. However, there's one situation in which a technical issue might make it difficult to preserve this style. From the scripting point of view, you'd really like to allow the user to write something like the following:

```
attach <document-list> to <mail-message-target>
```

The problem is that OpenDoc requires the target to be in the direct parameter. In the preceding script, the target is in the **to** parameter, not the direct parameter. To make this compatible with OpenDoc, you'll need to change the **attach** verb to **attach to** and swap the direct parameter and the **to** parameter, like this:

```
attach to <mail-message-target> documents <document-list>
```

Help your users figure out which objects to use with a verb.

Due to limitations in the 'aete' resource, there's no provision for indicating which Apple events can act on which objects. The AppleScript compiler will accept any combination of verbs and objects, even though some of these combinations have no meaning to your application and will result in runtime errors. To help your users determine which objects work with which verb, you can use the following trick.

Define the parameter's type as an enumeration instead of an object specifier. Use a # as the first character of the 4-byte ID for the enumeration. Then define the enumerators as the object classes that are appropriate for the event. You can use the same enumeration for more than one event; you can define different enumerations with different sets of object enumerators for different events; and you can even indicate the same object class in more than one enumeration. For example, instead of

```
close reference
```

a dictionary entry incorporating this technique would read

```
close window|connection|folder
```

This entry indicates to the user that the only object classes that make sense for the **close** command are **window**, **connection**, and **folder**.

OTHER TIPS AND TRICKS

Think carefully about objects versus properties.
Often, most of the work in a script is accomplished through creating objects and setting and getting properties, so use properties liberally. Be mindful that in certain cases, what initially might seem to be good candidates for objects might, on more careful examination, be represented as properties of another object, particularly when there's only one of such an object in your application. On the other hand, don't make something a property just because there's only one of it (such as a single object class belonging to an application or a containing object).

It's not always clear which is the better way to go — object or property. Some examples may help you understand how to decide this. Certain Finder objects have properties but are themselves properties of the application or the desktop container. The selection, an object of the abstract "selection-object" class, has properties such as the selection's contents. However, the selection-object class is never actually used in scripts; **selection** is listed as a property of the application and other selectable objects, so that a script writer doesn't need to form an object specifier, and the class name can be used as the object itself ("selection" instead of "selection 1").

As another example, a tool palette, which would normally be an object class, might be one of several objects of the palette class, or it might be better listed as a property of the application. This would depend on whether you had several named palettes (palette "Tools," palette "Colors") or wanted separate identifiers for each palette (tool palette, color palette). It could also depend in part on whether there were properties (and perhaps elements) of the palettes. In this particular case, using the **tool palette** and **color palette** properties is more localizable than including the name of the palette in the script. If you translate the program into some other language, it's a fair bet that the tool palette won't be named "Tools" anymore. However, your 'aete' resource will have been localized and thus **tool palette** will be transformed into the correct name for the object.

Try to be careful when deciding whether to make something a property or an object — users can end up writing

```
<property> of <property> of <object>
```

or even

```
<property> of <object> of <property> of <object>
```

and may become confused by real objects that appear to be datalike or that normally would be elements but are presented as properties. Make something a property only when it's meaningful rather than for convenience; otherwise, the concept of an object model hierarchy becomes eroded.

Whether something is a property or an object really depends on the specifics of your application. Still, in a large number of cases, objects are things that can be seen or touched, while properties are characteristics of the objects or the application. A good rule of thumb is: If the item in question is a characteristic of something else, it's probably a property.

Use inheritance to shrink your 'aete'.
If you've got a large 'aete' resource, or large groups of properties used in multiple classes, you can reduce the size and repetitiousness of your 'aete' by defining those sets of properties in an abstract or base class. Then classes that include those property definitions can include an inheritance property, with the ID code 'c@#^' (pInherits), as their first property. The human name for this property should be **<Inheritance>** (be sure to include the angle brackets as part of the name). The inclusion of this property will indicate to the user that this class inherits some or all of its properties from another class.

As an example, in QuarkXPress, several of the object classes have a large number of properties. Without inheritance, there would have been up to a hundred properties in the dictionary's list of properties for some of the classes! By creating abstract base classes in the 'aete' (defined in the application's Type Definitions suite) and inheriting from these, the application uses the same sets of properties (some quite large) in several different classes. The size of the 'aete' resource was reduced from 67K to 44K, and the lists of properties for many of the classes were reduced to just a few, including the inheritance property.

On the other hand, because this method produces a hierarchy that's smaller but more complex (and therefore slightly more confusing), I recommend using it only in situations where inheritance applies to more than one class. If you plan to use inheritance in only one place in your 'aete', or if your 'aete' isn't particularly large, it's probably better just to repeat all the properties in each class without using inheritance.

Be cautious when you reuse type codes.
If you use the same term for more than one "part of speech" in your dictionary, use the same 4-byte code. For example, if you use **input** as a parameter, again as a property, and later as an enumerator, use the same type code for each of the various uses.

By contrast — and this is very important because it's the single most common source of terminology conflicts — don't use the same type code for more than one event, or more than one class, and so on. If you do, AppleScript will change the script to show the last event or class defined with that code, changing what the user wrote in the script. This is usually not the desired effect, unless you specifically want synonyms.

If you do want synonyms, you can create them this way. For instance, in HyperCard the term "bkgnd field" is defined before "background field." The former can be typed and will always be transformed into the latter at compile time, so that the latter is always displayed. Just be careful not to have the script appear to change terminology indiscriminately — it's unsettling to the user.

The section "ID Codes and the Global Name Space" later in this article discusses additional considerations having to do with type codes.

Avoid using *is* in Boolean property and parameter names.
Because **is** can be used to mean "=" or "is equal to," and because it's a reserved word, you should avoid using it in human names for properties and parameters, such as **is selected**, **is encrypted**, or **is in use**. It's better, and less awkward, to use **selected**, **encrypted**, and **in use** or **used**. In a script, writing

```
if selected of thing 1 then ...
```

or

```
tell thing 1
   if selected then...
end tell
```

is better than writing

```
if is selected of thing 1 then ...
```

or

```
tell thing 1
   if is selected then ...
end tell
```

However, it's OK to use **has** or **wants** (which have none of the problems presented by **is**), as in

```
if has specs then ...
```

or

```
set wants report to true
```

When you name your Boolean parameters, keep in mind that AppleScript will change **true** and **false** to **with** and **without.** If the user writes

```
send message "Fred" queuing true
```

it compiles to

```
send message "Fred" with queuing
```

Control the number of parameters.
Sometimes you may find yourself implementing a verb that contains lots of options, for which you might be tempted to make separate Boolean parameters. When the number of parameters is small, it looks good to be able to say "with a, b, and c." Excessive use of this technique, however, can lead to unwieldy dictionary entries for these events with long lists of parameters.

There are two solutions to this:

- Make a parameter or parameters that accept a list of enumerators for the option or set of options.

- Break the command into separate commands with more focused functionality, reducing the number of options for each event.

For example, suppose a statistics package creates a single command to perform any type of analysis with lots of parameters, like this:

`analyze <reference>`	75 Boolean parameters indicating various analysis options

It would be better to split the analysis capability into multiple commands, followed by small groups of Boolean parameters, forming a suite, such as

`cluster <reference>`	small number of Boolean parameters indicating clustering options, or list of enumerators
`correlate <reference>`	small number of Boolean parameters indicating correlation options, or list of enumerators
`fit curve <reference>`	small number of Boolean parameters indicating curve-fitting options, or list of enumerators

and so on.

Use replies meaningfully.
In your dictionary, including a reply in an event's definition helps the user understand the behavior of an application-defined command and its role in the communication between a script and your application. However, you shouldn't include a reply definition if the only possible reply is an error message (except in the rare case where the error message is a normal part of the event's behavior).

When you return an object specifier as a reply, as in the case of the **make** command, it's up to you to decide which reference form to use. *Reference forms* (the various ways objects can be described in a script), also known as *keyforms*, include the following:

- name ("Fred", "Untitled 1")
- absolute (first, second, middle, last)
- relative (after word 2, behind the front window)
- arbitrary (some)
- ID (ID 555)
- range (4 through 6)
- test (whose font is "Helvetica")

For more information on reference forms, see *Inside Macintosh: Interapplication Communication* and the *AppleScript Language Guide*.•

Most scriptable applications to date implement the absolute reference form, such as **window 1**, as the reply to a **make** command. If your users are likely to change the position of this object during a script, you might consider using the name form instead. When you absolutely want a unique value, reply with the ID form, as in **window ID -5637**. The ID reference form ensures a unique value but usually means much less to the user.

Deciding which reference forms to use for object specifiers comes into play in applications that are recordable, as well.

APPROACHES TO RECORDING COMMANDS

If your application will be recordable, take note. Some early adopters of AppleScript recordability assumed that their users would only record an action and play it back to see an example of how to script it. Their early scripting implementations were done quickly, often without supporting the object model. Later they realized that users would actually write scripts, sometimes from scratch, using the dictionary as their guide. As a result, most have redone their implementations to clean them up or use the object model. Don't use recordability as an excuse to take the easy route and implement quickly. You'll end up wanting to redo it later, but you won't be able to because your installed base will be too large. Instead, implement the object model the first time.

There are two approaches to recording commands. One approach is to send something as close as possible to what the user would write to the recorder. This isn't necessarily a mirror image of the user's actions but produces recorded statements that more closely resemble what a user will write.

```
open folder "Goofballs" in disk "Razor"
```

The other approach is to duplicate the actions of users. This is the method used in the Scriptable Finder. In this method, what's recorded is that the user makes a selection and then acts on that selection.

```
select folder "Goofballs" in disk "Razor"
open selection
```

In the first case, the recorded statement helps the user understand how to write the command (my personal favorite). In the other case, there's a relationship between what the user did and what was recorded. Either method is useful — it depends on your objectives.

As is the case with returning object specifiers as replies (discussed above), you decide which reference forms to use for object specifiers that get recorded.

ID CODES AND THE GLOBAL NAME SPACE

One of the areas of greatest confusion among AppleScript developers is AppleScript's global name space and its implications for choosing ID codes for properties and enumerators. In this name space are all the terms used in all the scripting additions installed on a user's computer (see "If You're Writing a Scripting Addition . . .") and all the terms defined by AppleScript as reserved words. Properties and enumerators must have either unique or identical codes, depending on the situation. (Events, parameters, and classes that are defined within an application's dictionary aren't affected by this requirement.)

As noted earlier, you can reuse terms for different "parts of speech" — for example, for a parameter, a property, and an enumerator — but then you must use the same 4-byte ID code. By extension, if the term you want to use for a property or an enumerator is defined in the global name space, you *must* use the 4-byte code already defined there. For example, if you want to use the property **modification date**, you must use the code 'asmo', which is defined in the File Commands scripting addition. This applies across different parts of speech, so if, for instance, the term you want to use for a parameter is already defined in the global name space as a property, you must use the same code. If you use a different code, scripts that include your term may not compile, or they may compile but send the wrong code to your application when executed.

IF YOU'RE WRITING A SCRIPTING ADDITION . . .

Scripting additions (otherwise known as *osaxen*, the plural of *osax*, for OSA extension) add new core functionality to AppleScript by extending the AppleScript language. If you're writing a scripting addition, either for general purposes or for use with a particular application, you should be aware of a growing problem: the increasingly crowded name space for commands. When the number of additions was small, it was simple; each command (term) generally had only one usage. Now the situation is beginning to get out of hand.

The problem stems from three issues:

- Unlike applications, which generally go through a fairly significant development cycle, many osaxen have been written by programmers who aren't commercial application developers. As a result, there tend to be a great many more osaxen than scriptable applications.
- The name space for osax terminology is global in the sense that these gems are accessible from any script running on your computer. You might think of all the osax dictionaries being lumped together as though they were a single large application's dictionary (really a "system-level" dictionary). So when two or more osaxen use the same terms in slightly (or radically) different ways, trouble abounds. Only one of them will capture AppleScript's attention, and you, the osax author, can't control which will win out.
- If an application command is named the same as an osax command, the application command will be invoked inside a **tell** block, while the osax will be invoked outside the **tell** block. On the other hand, an osax command executed inside a **tell** block for an application that doesn't define the same command name will invoke the osax. Users writing scripts will undoubtedly make errors.

It's impossible to completely avoid every term used in every application, but where possible, try not to use terms that are likely to be used by application developers. Remember that a user may load up a computer with any number of osax collections, without realizing that there are four different **rename file** osaxen among the horde (or should I say herd?).

In addition, remember that if, for example, you define an **open file** command as an osax, the command

```
open file "curly"
```

is ambiguous. A user might want the Open event

```
open (file "curly")
```

or an osax command,

```
(open file) "curly"
```

Again, be extra careful when defining system-level terms.

A different problem exists in the special case where a set of osaxen is marketed for use with a special application, such as plug-ins or database connectivity. In this case, you should name your commands so that they are unmistakably associated with their host application. One possible solution is to begin the command names with a prefix indicating that they should only be used with the particular application.

Conversely, if you make up a new 4-byte ID code for your own property or enumerator, you need to take reasonable precautions to avoid using a code that corresponds to another term in the global name space. If you don't use a completely new code, you can't be sure which term is represented by that code in scripts that contain the code. So, for example, you shouldn't use the code 'asmo' unless you're referring to the **modification date** property.

How can you identify potential conflicts? One way is by using a script editor, MacsBug (with the **aevt** dcmd and the **atsend** macro), and the templates on the AppleScript Developer CD, notably the templates for the Apple Event Manager traps. Together, these tools enable you to catch an Apple event as it's sent and to examine it. Here's what you do:

1. Use the Formatting menu item in the editor to set the colors of the AppleScript styles so that you can see whether a term parses as an application-defined term or as a script-defined variable.

2. Type in your desired terminology and compile.
3. If it parses as a script-defined variable, it's free and you can use it with your own unique code to represent your own term. If it parses as an application-defined term, go on to the next step.
4. Break into MacsBug, type "atsend," and go. Execute the script, and the code for the property or enumerator will be displayed. You can then use this term in a manner consistent with standard terminology or definitions in scripting additions — the appropriate ID code will be generated by AppleScript. You must still include this term, along with the ID code you just discovered, in your 'aete' resource so that users will see the term in your dictionary. Then things will still work if the scripting addition that defines the term is subsequently removed.

IT'S NOT TOO LATE TO CLEAN UP YOUR ACT

Let's say you took a first stab at scriptability, implemented it in your application, and shipped it. Perhaps you did the expedient thing and didn't implement the object model. Or maybe you implemented totally new terms in your dictionary. Don't be afraid to redo some of your scripting implementation — it's still early enough in the scripting game to clean up your vocabulary or to go the distance and support the object model. It's *much* better to do it now, when there are only 50 or 100 people struggling to script your application. The overwhelming majority of your users will breathe a sigh of relief and thank you profusely for making their lives easier, even if they have to modify some of their existing scripts.

Two well-known developers have each recently done a relatively full scripting implementation and have indicated to their users that this is the first version, that some of it is experimental and is likely to change. A number of others have retraced their steps, rethinking their approach, and on occasion switched to object model support. I'll give two examples of applications where changing a scripting implementation made a significant difference.

EUDORA: CLEANING UP VOCABULARY

As one of the most widely distributed applications in the history of the Macintosh, Eudora by Qualcomm is used by a vast number of people to manage their Internet mail. Eudora originally used completely nonstandard terms. For example, this script created a new message and moved it to a specific mail folder:

```
tell application "Eudora"
   CreateElement ObjectClass message InsertHere mailfolder "Good stuff"
   Move message 1 InsertHere mailfolder "Other stuff"
end tell
```

This was an easy cleanup job, involving mostly just changes to the dictionary. Standard human terms were substituted for Apple event constructs, as can be seen in this script that now accomplishes the same thing as the preceding script:

```
tell application "Eudora"
   make new message at mail folder "Good stuff"
   move message 1 to mail folder "Other stuff"
end tell
```

Your terms don't have to be quite this far afield for you to consider a scripting facelift.

STUFFIT: SWITCHING TO THE OBJECT MODEL

By contrast, in the case of StuffIt from Aladdin, the developer revamped the application, replacing a non–object model implementation with one that supports the object model. This revision produced a dramatic increase in the ease of scriptability.

Here's a synopsis of the original implementation:

- Required suite: OpenApp, OpenDocs, PrintDocs, QuitApp
- StuffIt suite: Stuff, UnStuff, Translate, Copy, Paste, Clear, Get Max Number of Archives, Get Current Number of Archives, Stack Windows, Tile Windows, Get Version
- Selection suite: Select, Select All, DeSelect All, Select By Name, View Selected Items, Rename Selected Items, Delete Selected Items, Get Selected Count, Get Selected Name . . .
- Archive suite: New Archive, Create New Folder, Open Archive, Close Archive, Verify Archive, Get Archive Pathname, Get Archive Name, Set/Get Archive Comment, Set/Get Archive View, Stuff Item, UnStuff Item, Change Parent, Save
- Item suite: Get Item Count, Get Item Type, Get Item Name (and 14 others beginning with "Get Item"), Rename Item, Delete Item, Copy Items, Move Items

Notice the redundancy of Set, Get (more than 20 occurrences), Rename, Delete, Stuff, UnStuff, and Select. Also, notice that the command names look much like Apple event names. It was extremely hard to figure out how to script this application.

Once the object model was implemented, the scheme became a lot simpler:

- Required suite
  Events: open, print, quit, run
- Core suite
  Events: make, delete, open, and so on (the 14 main events)
  Classes: application, document, window
- StuffIt suite
  Miscellaneous events: cut, copy, paste, select
  Custom events: stuff, unstuff, view, verify, segment, convert
  Classes: archive, item, file, folder
- Type Definitions suite
  3 special record types used as property types in other classes

Each of the classes has a multitude of properties, where most of the action takes place. All the redundancies have been removed (the verbs can be remembered and used naturally), and statements can be written that resemble those written for other applications. The entries in the Type Definitions suite are record types used for properties. The result of this redesign is that the dictionary is now smaller and more understandable. A script to access all the items in an archive that was originally 68 lines long is now only 20 lines!

THE JOURNEY BEGINS

Making your application scriptable is an art. Think of AppleScript as a living, growing human language. As you've seen, there are standard terms and object model constructs that you can use when designing your application's scripting implementation, for those capabilities that are common to many or all applications.

In the end, though, a unique treatment is usually necessary to fully express the particular capabilities of each application, and your scripting implementation should be carefully constructed accordingly.

I hope this article has convinced you to do the following:

- Make AppleScript *your* application's language. Remember that AppleScript isn't just for programmers — many users will want to write and record scripts to control your application.
- Develop a sense of style. Consider the nature of what your users will end up writing in their scripts. "Clean and elegant" (like a user interface) will serve your users well. Use human terms that can be easily understood by a nonprogrammer.
- Strive for consistency. Follow the conventions, suggestions, and general guidelines outlined here, for the sake of semantic consistency across applications.
- Choose your terms carefully. Consider whether and how the terms you use in your vocabulary will affect the name space for AppleScript.

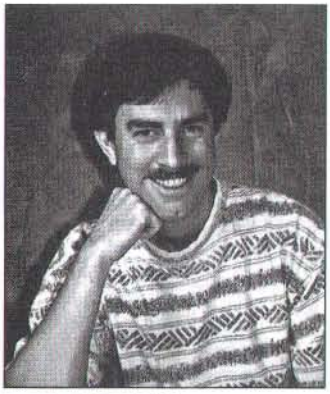
On the other hand, if you aren't comfortable designing a semantic vocabulary or if you're having trouble formulating a clear picture in your mind of a natural-language sentence structure, don't attempt to do it yourself. As in the case of graphic and interface design, it might be better to engage the services of an expert.

If you do undertake designing a scripting implementation yourself, you'll find it to be a rewarding experience, one that can enable your users to accomplish things never before possible. Happy implementing!

RESOURCES

- *Inside Macintosh: Interapplication Communication* (Addison-Wesley, 1993), Chapters 3 through 10. (*Inside Macintosh* Volume VI is *not* recommended.)
- "Apple Event Objects and You" by Richard Clark, *develop* Issue 10.
- "Better Apple Event Coding Through Objects" by Eric M. Berdahl, *develop* Issue 12.
- *Apple Event Registry: Standard Suites*, available on this issue's CD or in print from APDA.
- AppleScript Software Development Toolkit, available from APDA.
- *AppleScript Language Guide* (Addison-Wesley, 1993). Also in the AppleScript Software Development Toolkit.
- The Webster Project. This master database, containing terms used in scriptable applications and scripting additions, assists in resolving naming collisions across applications and serves to regularize the common terms used by applications of different types. I'm designing and implementing this; contact me at AppleLink MAIN.EVENT for more information.

Thanks to our technical reviewers C. K. Haun, Don Olson, and Jon Pugh, and to Michael Bayer.•



PRINT HINTS

Writing QuickDraw GX Drivers With Custom I/O and Buffering

DAVE HERSEY

One of the great features of QuickDraw GX is that it provides the printer driver developer with default implementations of commonly used routines. For example, just by specifying a few parameters in your driver's 'comm' (gxDeviceCommunicationsType) resource, your printer driver can connect to a printer either serially or through the Printer Access Protocol (PAP). You don't need to write a single line of communications code!

Another powerful feature of QuickDraw GX is that you can ignore the default implementations of printer driver routines and write your own routines instead. This feature enables you to tailor your printer driver so that it can accommodate unique situations. The ability to modify bits and pieces of the printing system is especially useful when it comes to writing printer drivers with custom communications code or buffering routines.

In general, to create custom communications code, you configure your driver's 'iobm' (gxUniversalIOPrefsType) resource, create a "not connected" 'comm' resource, and then override certain QuickDraw GX messages. For SCSI printers, however, you don't need to create the "not connected" resource, because a SCSI format of the 'comm' resource is already defined. We'll talk more about when you would want to use custom communications code, and how to write it, later in this column.

Also covered is how to write custom buffering routines. You may want to use custom buffering if, for example, you already have code that you want to use or you want to increase printer performance by taking advantage of a hardware buffer that you have available.

On this issue's CD, you'll find a sample printer driver called CustomWriter that illustrates how to implement "not connected" custom I/O and buffering. In addition, there's a sample LaserWriter IISC printer driver that shows how to create custom I/O code for a SCSI printer.

CUSTOM I/O — WHO NEEDS IT?

The default communications code in QuickDraw GX handles asynchronous communications for serial and PAP printers and QuickDraw GX shared printers. Even so, you may want to override this code in some cases, such as if your printer communicates using a protocol that QuickDraw GX doesn't support (like 200 Kbits/second serial), or if you have your own PAP code that you'd like to continue using.

QuickDraw GX also supports the special cases of "not connected" printers and SCSI printers. If you're writing a driver using either of these two types of connections, you'll need to write some custom I/O code. In fact, the "not connected" communications method is provided specifically for the developer writing a driver containing custom communications code. What does this type of communications method do? In the default implementation, nothing at all. In a minute, you'll see how to use this to your advantage.

The only SCSI support currently built into QuickDraw GX handles filling out the Chooser list with your devices' SCSI addresses and saving updated 'comm' resources for any desktop printers that are created. Otherwise, QuickDraw GX doesn't actually open connections or try to send commands, such as SCSIRead or SCSIWrite, to the printer. SCSI printers usually have unique command sets, and trying to provide a generic mechanism to support all of these devices is unrealistic. As a result, you must provide your own communications code if you're writing a SCSI driver.

Finally, if your device is connected through a hardware interface that QuickDraw GX doesn't provide default support for (such as a NuBus™ card), you'll need to provide all of the communications code for your driver.

HOW TO GET STARTED

The first step in writing a driver with custom communications code is to configure your driver's 'iobm' resource. This is a very easy (and very critical) exercise.

DAVE HERSEY (AppleLink HERSEY) left Apple's Developer Technical Support (DTS) group about six months ago to join the Print Shop software development group. He now fixes the QuickDraw GX bugs that he reported while in DTS, and works on QuickDraw GX 2.0 — the "knock your socks off" release.•

'iobm' stands for "Input/Output and Buffering preferences." So what does the "m" stand for? Great question. As it turns out, if you set up this resource incorrectly, it becomes an "I/O BooM" resource. (The system crashes.) The "m" is silent as long as the resource is set up correctly.•

The 'iobm' resource tells QuickDraw GX how your driver wants its communications and buffering environment set up. This resource has the following format:

```
type gxUniversalIOPrefsType {
   longint   standardIO = 0x00000000,
             customIO = 0x00000001;
   longint;     // number of buffers to allocate,
                //  0 = none
   longint;     // size of each buffer
   longint;     // number of IO requests that can
                //  be pending at any one time
   longint;     // open/close time-out in ticks
   longint;     // read/write time-out in ticks
};
```

The 'iobm' resource was described in the *develop* Issue 20 Print Hints column about QuickDraw GX buffering. Rather than reiterate that information here, we're going to briefly focus on the first three fields of the resource.

The first item in the 'iobm' resource (standardIO or customIO) tells QuickDraw GX whether you want to use its built-in communications code. You must specify customIO if you want to use your own custom I/O code. When customIO is specified, QuickDraw GX won't go through the overhead of initialization and data allocation for the internal communications routines; as a result, you *must* override certain messages, as described in a following section. When you specify customIO, the last three longint fields of this resource are ignored.

The two fields in the 'iobm' resource that follow the I/O type field indicate the number and size of the buffers your driver would like QuickDraw GX to create. Note that you can use QuickDraw GX's built-in buffering even if you're writing your own communications code. If, however, you're creating and disposing of your own buffers, you should set the "number of buffers" field to 0, so that QuickDraw GX won't waste time and memory allocating buffers that are never used. For code that communicates synchronously, multiple buffers don't improve performance, so you should set this field to 1.

Later in this column we'll take a closer look at what's required to create and manage your own I/O buffers.

WHEN "NOT CONNECTED" MEANS "CONNECTED"

Unless you're writing custom I/O routines to support a SCSI printer, you'll want to create a "not connected" 'comm' resource for your driver. Below is the declaration of a 'comm' resource for the "not connected" case.

For the full description of a 'comm' resource, see *Inside Macintosh: QuickDraw GX Printing Extensions and Drivers.*•

```
type gxDeviceCommunicationsType {
   unsigned longint = 'nops';
};
```

There's not a whole lot to it, is there? When you specify customIO in your 'iobm' resource, QuickDraw GX never does anything with your desktop printer's 'comm' resources other than examine the first longint. So, all sorts of possibilities become apparent. As long as that first longint is 'nops', you can extend the definition of this resource to suit your needs. Whether to change the definition of the resource in the PrintingResTypes.r interface file or not is up to you. Instead, you could just resize the resource when you update it at desktop printer creation time, as we'll discuss momentarily.

CUSTOM I/O — THE MESSAGES

When you supply your own I/O routines, there are several messages that you need to override. Some of these messages will always need to be *totally* overridden, meaning that your overrides for these messages should never forward the messages. Other messages should be *partially* overridden, in which case the message is forwarded at some point in your override code.

Now we'll look at the messages you need to override. (Table 1 summarizes these messages and the ones to override for custom buffering.) If you want more information on writing message overrides, see Sam Weiss's article, "Developing QuickDraw GX Printing Extensions," in *develop* Issue 15, and *Inside Macintosh: QuickDraw GX Printing Extensions and Drivers.*

Always override (partially)

- GXOpenConnection
- GXCloseConnection
- GXCleanupOpenConnection
- GXWriteData

When you override these messages, you should first forward the message, then execute your added code. Your overrides for the first three messages should contain code to open and close a connection to your device.

Table 1. Overriding QuickDraw GX messages

When to Override	Custom I/O	Custom Buffering
Always override (partially)	GXOpenConnection GXCloseConnection GXCleanupOpenConnection GXWriteData	GXOpenConnection GXCloseConnection GXCleanupOpenConnection
Always override (totally)	GXDumpBuffer	GXBufferData GXWriteData
Usually override (partially)	GXDefaultDesktopPrinter GXChooserMessage	
Sometimes override (totally)	GXFreeBuffer	

GXCloseConnection is sent to close a connection if no errors occur during the device communications phase of printing; GXCleanupOpenConnection is sent if an error does occur during this time. The goal for both of these overrides is to "undo" any data allocation or initialization that occurred in the GXOpenConnection override. Often, your GXCleanupOpenConnection message override can simply execute the same code as your GXCloseConnection override.

The GXWriteData override should forward the message (with a nil pointer and a length of 0) to flush any data that's buffered, and then send the data to the printer.

Always override (totally)

- GXDumpBuffer

Your override for this message should execute code that sends the indicated data to your printer. When this message is sent, a connection to your device will already have been established through the successful execution of your GXOpenConnection override. The GXDumpBuffer message is used to send data to the printer whenever an I/O buffer becomes full.

Usually override (partially)

- GXDefaultDesktopPrinter
- GXChooserMessage

When QuickDraw GX creates a desktop printer, it stores in it a 'comm' resource that specifies how to communicate with the printer. By default, this 'comm' resource is just a filled-in copy of one of your driver's 'comm' resources. Depending on the setting in the Chooser's "Connect via:" menu, the 'comm' resource is updated with information about the printer, such as the selected serial port, SCSI address, and network address for an AppleTalk printer. If your driver uses a "not connected" 'comm' resource (as described earlier), it will be copied verbatim, without updated information about the selected printer. As a result, you might need to step in and fill out the resource yourself.

To update the 'comm' resource, you need to override the GXDefaultDesktopPrinter message as shown in Listing 1. Here we forward the message so that QuickDraw GX completes creation of the desktop printer; then we retrieve the 'comm' resource from the desktop printer, update it, and replace the old version with the updated version.

When you update the 'comm' resource, you need to know which printer the user selected, as well as its addressing information and so forth. You can find this information by overriding GXChooserMessage, which is sent by the GXHandleChooserMessage API call. In this override, possibly with some help from your Chooser PACK's LDEF, you can determine the relevant information about the selected printer.

For example, you can store this information in a column of cells that's appended to the printer list. Or you can store it in the list record's userHandle or, by using PC-relative addressing, in a data storage area following your jump table. When you retrieve this data in your GXChooserMessage override, simply store it using one of QuickDraw GX's global data functions for printing. Finally, retrieve the information from your GXDefaultDesktopPrinter override and store it in the desktop printer's 'comm' resource.

It's important to note that you can't use any of the functions GetMessageHandlerInstanceContext, SetMessageHandlerInstanceContext, GXGetJobRefCon, GXSetJobRefCon, and NewMessageGlobals for the example in Listing 1, because the GXChooserMessage and GXDefaultDesktopPrinter messages are sent to two different message handler instances. Therefore, you should use GetMessageHandlerClassContext,

SetMessageHandlerClassContext, or some other method that works across message handler instances.

Sometimes override (totally)

- GXFreeBuffer

If your communications code runs asynchronously, you must override GXFreeBuffer so that QuickDraw GX can tell when operations on a buffer have completed. The GXFreeBuffer message is sent to make sure that all the data in the buffer has been processed before the buffer is used again. When GXFreeBuffer returns, the indicated buffer is ready to accept more data. An override for this message should loop (calling GXJobIdle) until I/O on the specified buffer is complete, and then return.

USING YOUR OWN BUFFERING SCHEME

At this point, we've discussed everything that's needed to handle your own custom I/O code. Now we'll take a quick look at what's required if you want to create and maintain your own buffers, instead of using those that the default implementation provides.

First things first. Go back to your 'iobm' resource, and set the number of buffers to 0. This tells QuickDraw GX not to waste time and memory allocating buffers that you aren't going to use.

When you implement your own buffering scheme, you can use any sort of internal representation for your buffers that you want to. However, since some of the buffering messages take a pointer to a gxPrintingBuffer, you'll need to use that format for passing your buffers between certain messages. But as far as the actual buffer structures go, you can use a handle, a linked list, or any other configuration that's convenient or necessary to use.

To support custom buffering code, you'll need to override the following messages.

Always override (partially)

- GXOpenConnection
- GXCloseConnection
- GXCleanupOpenConnection

Listing 1. Updating a 'comm' resource when a desktop printer is created

```
OSErr MyDefaultDesktopPrinter (Str31 dtpName) {
   OSErr     anyErrors;
   Handle    theCommResource;

   // Forward the message so that the desktop printer is created.
   anyErrors = Forward_GXDefaultDesktopPrinter(dtpName);
   nrequire(anyErrors, Abort);

   // Load the data for the 'comm' resource that was stored in the desktop printer.
   anyErrors = GXFetchDTPData(dtpName, gxDeviceCommunicationsType, gxDeviceCommunicationsID,
                              &theCommResource);
   require_action(theCommResource != nil, Abort, anyErrors = resNotFound;);

   // Update the 'comm' data with info about the selected printer, and store the updated copy
   // back in the desktop printer.
   MyUpdateCommResource(theCommResource);
   anyErrors = GXWriteDTPData(dtpName, gxDeviceCommunicationsType, gxDeviceCommunicationsID,
                              theCommResource);

   // Finally, dispose of the handle we received from GXFetchDTPData. It's a detached resource
   // handle, so DON'T USE RELEASERESOURCE!!
   DisposeHandle(theCommResource);

Abort:
   return anyErrors;
}
```

The partial overrides for these messages should forward the messages and then allocate or dispose of your internal buffer structures. If you're using custom I/O, you already provide overrides of these messages. In that case, simply add this new code to the existing overrides.

Always override (totally)

- GXBufferData
- GXWriteData

Provide an override of GXBufferData that stores the passed data in your next available buffer. If a buffer becomes full, call Send_GXDumpBuffer. Before you attempt to add data to this buffer again, call Send_GXFreeBuffer to make sure that all of the buffer's data has been sent to the printer.

Your override for the GXWriteData message should flush all data from your buffers and then immediately send the passed data to the printer. To do this, call Send_GXDumpBuffer on all buffers, followed by Send_GXFreeBuffer on all buffers. If you're performing custom I/O, just add this code to your existing override.

You may wonder why you don't need to override the GXDumpBuffer message when you perform custom buffering. Unlike the messages listed above, GXDumpBuffer takes a pointer to a gxPrintingBuffer. Whenever your code calls Send_GXDumpBuffer, you must pass data in a gxPrintingBuffer structure, regardless of the internal buffer representation that you're using. Since the buffered data is passed in the format that GXDumpBuffer already expects, there's no need to override the message.

DRIVE SAFELY

That's all there is to it. So, the next time someone asks, "Can you write QuickDraw GX printer drivers for my 200 Kbits/second serial typesetter, my SCSI copier/printer, and my NuBus-interfaced cutter plotter?" tell them, "Yooooooooou betcha!"

Thanks to Tom Dowdy, David Hayward, and Nick Thompson for reviewing this column.

An Object-Oriented Approach to Hierarchical Lists

The article "Displaying Hierarchical Lists" in develop *Issue 18 showed how to use the List Manager to build and display lists of hierarchical data with triangular "twist-down" buttons for expanding and collapsing sublists (similar to the ones the Finder uses for displaying and hiding the contents of folders in a list view). In this article, we take an object-oriented approach to implementing these and other custom lists, using the PowerPlant application framework by Metrowerks. Using subclass inheritance to build small classes on top of each other makes incremental development easy and straightforward.*

JAN BRUYNDONCKX

Recently, I found myself working on a project that needed hierarchical lists: a remote debugger for a network-based software distribution application. The product, FileWave, creates a "virtual disk" volume on the user's client machine and manages its contents remotely from a central server. The debugger, called TheRaven, can retrieve file and folder information from the client machine and display it in a Finder-like hierarchical view (see Figure 1).

Martin Minow's article "Displaying Hierarchical Lists" (*develop* Issue 18) was an excellent starting point, but Martin's implementation had some features that made it unsuitable for my particular application. Most important, Martin built his hierarchical lists completely in memory before displaying them — not very practical when working over a network. I could have modified Martin's code to remove that restriction, but the result wouldn't have been very clean. Since we were using the object-oriented PowerPlant application framework by Metrowerks, I decided to try to develop an object-oriented implementation for hierarchical lists.

One of the advantages of object-oriented programming is that it enables you to build up your implementation in incremental steps. PowerPlant's collection of small, independent classes can be combined to build new classes with rich features, providing a strong foundation for software development. And, of course, using PowerPlant gave me an opportunity to try out the great Metrowerks CodeWarrior programming environment.

JAN BRUYNDONCKX (AppleLink WAVE.BEL) works at Wave Research in Belgium, trying to create the killer application that will revolutionize software distribution across networks. When not peering at TMON windows and telling everyone how "interesting" they look, Jan can be found jumping off cliffs with a parasail. (If parasails had a real operating system, they wouldn't crash into trees — but they'd also be less fun!) Jan's idea of a holiday is hiking through the Sahara Desert or climbing mountains in Nepal. His favorite conversation topic at parties is the similarities between classical opera and hard rock.•

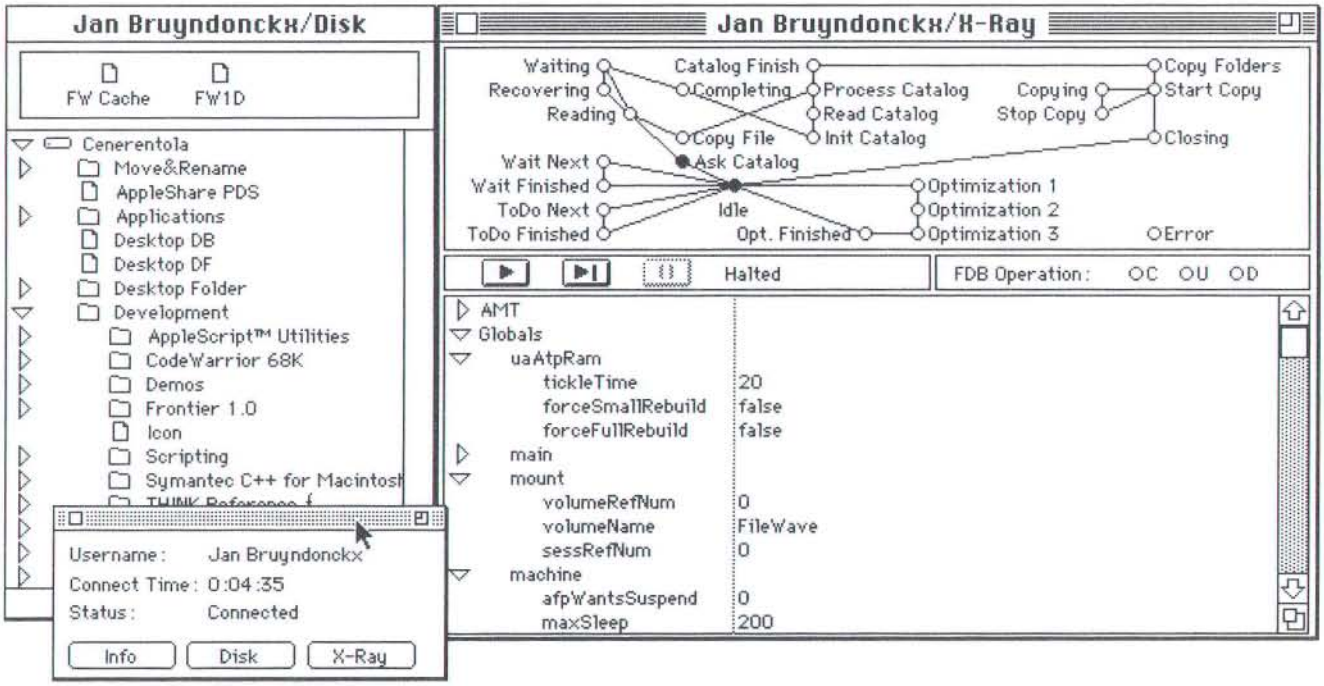


Figure 1. TheRaven

This issue's CD contains some of the results of my development efforts. On it, you'll find a collection of general-purpose classes for implementing lists with icons, hierarchical lists, and other useful possibilities. You can use these as a basis for developing more specialized subclasses of your own; the CD includes some examples of those, as well.

The CD contains two project files: one for creating a 680x0 application and one for the native PowerPC version. In both projects, only the main segment contains my own code; all the other source files are taken from the PowerPlant development framework. •

This article assumes that you understand the List Manager and how to use it, and that you have at least a casual acquaintance with object-oriented programming in general and C++ in particular.

BASIC BUILDING BLOCKS

In PowerPlant, everything that appears on the screen is a *pane*, an instance of the built-in class LPane. Like a view in MacApp, a pane can be anything from a plain rectangle to a scroll bar, a picture, or a radio button. A control is a pane, as is an icon button, a static text item, or a scrolling picture. Even LWindow, the class to which windows themselves belong, is a subclass of LPane.

Typically, a window consists of an instance of class LWindow with one or more subpanes derived from LPane. In our examples, our windows will have only one pane, an instance of PowerPlant's built-in class LListBox. This type of pane uses the Macintosh List Manager to display a list of objects. Each of our examples will define a new subclass of LListBox with additional or modified properties and behavior. All it takes to define such a class is to select an existing class, override its drawing method (and maybe a couple of others), and possibly create a new resource template.

EASY LISTS

Our first example implements a simple window showing the list of words "One" through "Five" (see Figure 2). This may not seem like a big deal, but it's a good illustration of the power of object-oriented programming.

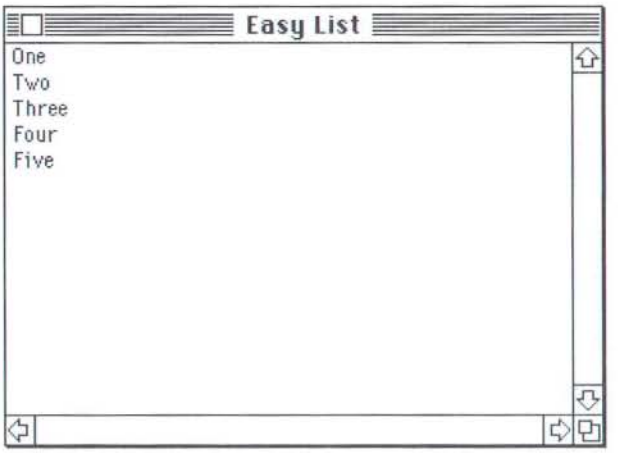


Figure 2. An easy list

If we started from scratch, how many lines of code would this application take? Well, we'd have to set up a menu, create a window, and then write an event loop to handle dragging, window resizing, and so on. Add in the List Manager calls, and we'd be lucky to do it all in fewer than 100 lines. With PowerPlant, all those details are handled for us by the predefined class LApplication. All we need to do is define a subclass, CListApp, with a menu command for creating our list window. One line of code in our subclass's ObeyCommand method suffices to create the window:

```
LWindow::CreateWindow(EasyList_PPob, this);
```

This invokes a static method of class LWindow to create the window from a template resource. EasyList_PPob is the resource ID; the exact description of the window is contained in the resource, isolated from the code itself.

The resource definitions (Listing 1) give the details on the window's structure and appearance. The familiar window template resource ('WIND') is accompanied by a PowerPlant object resource ('PPob') giving extra information on the window and the panes it encloses (see "'PPob' Resources"). The 'PPob' is simply a list of views and panes, each specified with the keyword ObjectData. Panes can be nested to any depth, with each new level delimited by the keywords BeginSubs and EndSubs. In our case, the window view encloses just one pane, representing the list box.

An object-oriented application framework like PowerPlant is so powerful that these two resources are all we need to describe our window and its list pane. With just one line of code to create the window, we get all the standard behavior for free: dragging and resizing the window, scrolling the list, and selecting items with the mouse. We can have multiple windows with the same list, and can use the List Manager for manipulations like adding or removing items.

But, of course, we won't stop there. In the following examples, we'll override the standard behavior by creating a series of subclasses. The resources in each case will be minor variations on the ones in Listing 1; the main difference is that we'll use a subclass instead of one of the standard classes.

Listing 1. Resources for easy list

```
resource 'WIND' (EasyList_WIND, purgeable) {
   {47, 17, 247, 317},
   documentProc,                        // standard window with size box
   visible, goAway,
   0x0,                                 // refCon
   "Easy List",
   noAutoCenter
};

resource 'PPob' (EasyList_PPob, purgeable) {{

   ObjectData {Window {
      EasyList_WIND,
      regular, hasCloseBox, hasTitleBar, hasResize, hasSizeBox, noZoom,
      hasShowNew, enabled, hasTarget, hasGetSelectClick,
      noHideOnSuspend, noDelaySelect, hasEraseOnUpdate,
      100, 100,                         // minimum width, height
      screenSize, screenSize,           // maximum width, height
      screenSize, screenSize,           // standard width, height
      0                                 // userCon
   }},

   BeginSubs {},
      ObjectData {ListBox {
         1001,                          // paneID
         {302, 202},                    // {width, height}
         visible, enabled,
         bound, bound, bound, bound,    // edges bound to superview
         -1, -1, 0,                     // left, top, userRefCon
         defaultSuperView,
         hasHorizScroll, hasVertScroll, hasGrowBox, noFocusBox,
         0, kGeneval0_Txtr,             // double-click msg, text traits
         textList,                      // LDEF ID
         {"One", "Two", "Three", "Four", "Five"}  // some sample data
      }},
   EndSubs {}

}};
```

CUSTOM LISTS

The previous example used the standard behavior of PowerPlant's built-in class LListBox. We can make our list much more attractive by adding an icon in front of each element. To do this, we'll define two new subclasses of LListBox.

Actually, one subclass would have been enough to do the job. But the most important thing I learned in my university software engineering courses was, "Be a toolsmith." Following this advice, I've chosen to define two subclasses instead of just one. The first, CCustomListBox, is a versatile, general-purpose tool that allows a list to hold any kind of data instead of just text. The items in the list can be structures of arbitrary size holding any kind of information we want. The CCustomListBox class includes methods for displaying this information easily and conveniently.

'PPOB' RESOURCES

BY AVI RAPPAPORT

Resources of type 'PPob' (PowerPlant object) represent objects that belong to PowerPlant's predefined class LPane and its derived subclasses. Their structure is fully described in the section "Creating Panes" (Chapter 9 in the August 1994 release) of the PowerPlant manual supplied on the CodeWarrior CD. Each 'PPob' resource describes an entire containment hierarchy — for example, an enclosing pane, then a scrollable "view," scrollers, and the window's buttons, list boxes, text fields, and radio button groups. You can also add new types to represent your own custom subclasses of LPane.

Object layering makes 'PPob' resources too complex for ResEdit's template mechanism, so you have to use Apple's Rez, Metrowerks' PowerPlant Constructor (provided on the CodeWarrior CD), or Mathemaesthetics' Resorcerer to edit them. The listings in this article are in Rez format. Note that Rez files must be compiled separately to be included in a CodeWarrior project, as the current version of CodeWarrior cannot compile them automatically.

Resorcerer provides a forms-based interface. To use it, copy the file PowerPlant Resorcerer TMPLs from the PowerPlant Resources folder to Resorcerer's Private Templates folder. The 'PPob' editor will be available the next time you start Resorcerer.

PowerPlant Constructor uses more of a point-and-click interface to display the user view for each object in a 'PPob' resource. You can edit the values in the Attributes palette and Field windows and view the results on the screen. For instructions on the specific menu items involved, see the *Constructor User's Guide* on the CodeWarrior CD.

Using PowerPlant and the 'PPob' resources together, you can create clean, standard interfaces for your programs, using the best of Apple's new technologies. This allows you to be more creative about the design of your programs and concentrate on adding new features to make the best possible applications.

The second subclass, CMyCustomListBox, is just a demo class to show off the capabilities of the first. It inherits the general behavior of CCustomListBox and specializes it to hold two pieces of information for each list item: an icon (actually, just the icon's resource ID) and a piece of text (see Figure 3).

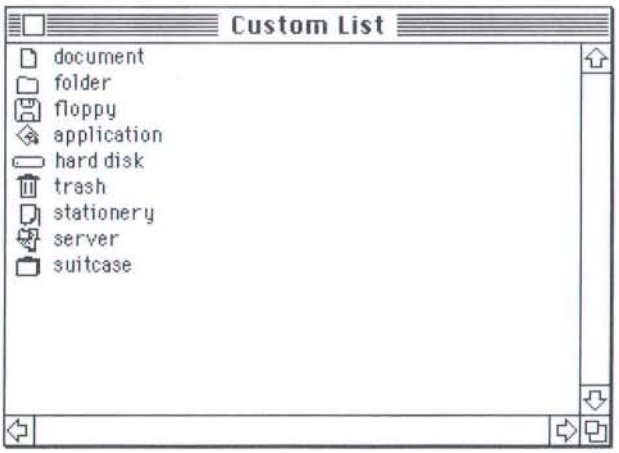


Figure 3. A custom list

CREATING A LIST

The template ('PPob') resource for our list pane has the same format as the standard one shown in Listing 1, but without the sample data (the strings "One" through "Five"), since we're now allowing the list to contain any kind of data instead of just text. This time, though, we want the window's list pane to be an instance of our custom class, CMyCustomListBox, instead of PowerPlant's predefined class LListBox.

The job of creating a new window at run time from a template resource is handled by a part of the PowerPlant system called the *reanimator*. We need to tell the reanimator to use our own creator method when creating the window's list pane from the template, in place of the standard one for class LListBox.

We establish the connection between our template resource and the creator method that will use it by assigning the template a unique *tag*. We then register the tag with the PowerPlant *registrar*, telling it to associate that tag with a particular creator method. We create the tag by adding the line

```
ClassAlias {'mlst'},
```

to our 'PPob' resource, before the definition of the list pane. (All we need is an alias, because the resource defining our custom class has the same structure as that of the standard LListBox class.) We then define a constant to represent this tag in our CMyCustomListBox class:

```
public:
   enum {
      classID = 'mlst'
   };
```

Now we can register the tag with the PowerPlant registrar as part of our application's initialization code:

```
URegistrar::RegisterClass(CMyCustomListBox::classID,
                  (ClassCreatorFunc) CMyCustomListBox::CreateFromStream);
```

(A convenient place to do this is in our application object's constructor method, CListApp::CListApp.) Later, when we use our template to create a new object —

```
LWindow::CreateWindow(CustomList_PPob, this);
```

— PowerPlant's reanimator will recognize the tag and will call the specified creator method, CMyCustomListBox::CreateFromStream, to create an instance of our class. We define the creator method as follows:

```
CMyCustomListBox* CMyCustomListBox::CreateFromStream (LStream *inStream)
   {
      return (new CMyCustomListBox(inStream));
   }
```

This simply passes along the parameter it receives, inStream, to the class constructor method, CMyCustomListBox::CMyCustomListBox. This method in turn calls the superclass constructor method, CCustomListBox::CCustomListBox, and then adds some further initialization of its own:

```
CMyCustomListBox::CMyCustomListBox(LStream *inStream) :
                                                CCustomListBox(inStream)
{
   // Additional initialization for class CMyCustomListBox
   ...
}
```

The extra initialization code calls the Macintosh List Manager to add cells to the list and initializes the contents of each cell. In some cases (though not in this example), it

might need to read in additional resource data. This is also the ideal place to initialize the new object's member variables.

CUSTOMIZING THE LIST DEFINITION PROCEDURE

The List Manager calls a *list definition procedure* to display each cell of a list on the screen (see *Inside Macintosh: More Macintosh Toolbox*, Chapter 4). The procedure is supplied as a code resource of type 'LDEF'. In our case, we want to keep the display code inside the application, so that we can define it as a method of our custom subclass. Figure 4 illustrates our scheme for accomplishing this.

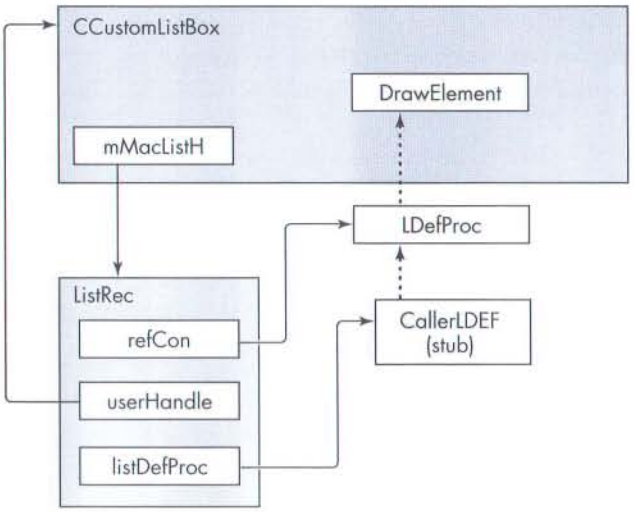


Figure 4. Customizing the list definition procedure

The LDEF that we supply to the List Manager is just a stub that calls the real one defined in our application. We use the refCon field of the list record to hold a "callback pointer" to the real definition procedure; the userHandle field holds a pointer back to the list object. The initialization method CCustomListBox::init sets all this up:

```
if (callerLDEFUPP == NULL)
   // Create UPP for LDEF callback.
   callerLDEFUPP = NewListDefProc(LDefProc);

// Put callback address in refCon.
(*mMacListH)->refCon = (long) callerLDEFUPP;

// Keep a pointer to self.
(*mMacListH)->userHandle = (Handle) this;
```

mMacListH is a member variable of LListBox containing a handle to the list record. First we create a universal procedure pointer (UPP) to our callback function, LDefProc, and store it in the list record's refCon field; then we save a pointer to the list object itself ("this") in the userHandle field. Finally, we load the stub LDEF from the resource file, save its handle in the listDefProc field of the list record, and make it unpurgeable from the heap.

Listing 2 shows the code of our callback function and the subsidiary methods it calls. The callback function, LDefProc, sets up the A5 world, looks in the list's data for the contents of the cell to be drawn, and calls the list object's member function

Listing 2. Custom list definition procedure for CCustomListBox

```
static pascal void LDefProc (short lMessage, Boolean lSelect,
                             Rect *lRect, Cell lCell,
                             unsigned short lDataOffset,
                             unsigned short lDataLen,
                             ListHandle lHandle)
// Custom list definition procedure for CCustomListBox.
// Called by the LDEF stub; returns control back to class method
// DrawElement to do the actual drawing.
{
   // Ignore init and dispose messages.
   if ((lMessage == lInitMsg) || (lMessage == lCloseMsg))
      return;

   // Set up application's A5 so that we can access global variables.
   long savedA5 = ::SetCurrentA5();

   // Get pointer to list object from userHandle field of list record.
   CCustomListBox *self = (CCustomListBox*) (*lHandle)->userHandle;

   // Get handle to cell data.
   Handle h = (*self->mMacListH)->cells;
   char saveState = ::HGetState(h);
   ::HLock(h);

   // Find and draw cell contents.
   void *lElement = (void*) (*h + lDataOffset);
   self->DrawElement(lMessage, lSelect, lRect, lElement, lDataLen);

   // Restore previous handle state and A5.
   ::HSetState(h, saveState);
   ::SetA5(savedA5);
}

void CCustomListBox::DrawElement (const short lMessage,
                                  const Boolean lSelect,
                                  const Rect *lRect,
                                  const void *lElement,
                                  const short lDataLen)
// Member function for responding to LDEF calls.
// Calls DrawElementSelf to draw a list element.
{
   switch (lMessage) {

      case lDrawMsg:
         ::EraseRect(lRect);
         if (lDataLen == 0)
            break;
         DrawElementSelf(lRect, lElement, lDataLen);
         if (!lSelect)
            break;
```

(continued on next page)

Listing 2. Custom list definition procedure for CCustomListBox *(continued)*

```
      case lHiliteMsg:
         ::InvertRect(lRect);
         break;
   }
}

void CCustomListBox::DrawElementSelf (const Rect *lRect,
                                      const void *lElement,
                                      const short lDataLen)
// Draw contents of a single list element on the screen.
// Default version just draws text; override for other types of data.
{
   ::MoveTo(lRect->left + 2, lRect->top + 9);
   ::DrawText(lElement, 0, lDataLen);
}
```

DrawElement to draw it. DrawElement clears the cell's rectangle to prepare for drawing, makes sure that the cell's contents aren't empty, and calls another member function, DrawElementSelf, to do the actual drawing. Then DrawElement checks its lSelect parameter to see whether to highlight the cell and, if so, inverts the cell's rectangle.

The default version of DrawElementSelf, defined in our CCustomListBox class, just draws a simple piece of text for the contents of a cell. More specialized subclasses, such as CMyCustomListBox, can override this method to draw other types of cell contents or to display them in different ways. (In unusual cases, a subclass might want to override the calling method, DrawElement — to redefine the way highlighting is done, for example.) Both DrawElement and DrawElementSelf are defined as virtual methods, ensuring that all calls are directed to the proper version for a particular class of list. This allows our application to support list boxes of many different kinds simultaneously, with each going through the same general LDEF, but ultimately calling its own specialized version of the drawing method.

As an example, Listing 3 shows the DrawElementSelf method for our class CMyCustomListBox. Each cell of the list displays both a small icon and a text label, as we saw earlier in Figure 3. The cell data in the List Manager's list record structure consists of the icon's resource ID (resource type 'SICN') and a Pascal-format string specifying the text:

```
typedef struct {
   short    iconID;
   Str255   name;
} MyCustomDataRec, *MyCustomDataRecPtr;
```

The DrawElementSelf method calculates a 16-by-16-pixel rectangle for the icon, plots it with CopyBits, and then draws the text. There's no need to override the DrawElement method, since the standard form of highlighting is all we need.

HIERARCHICAL LISTS

Our next example is a hierarchical "twist-down" list like those in Martin Minow's Issue 18 article. Our version lacks a few of the more advanced features of Martin's —

Listing 3. Drawing method for CMyCustomListBox

```
void CMyCustomListBox::DrawElementSelf (const Rect *lRect,
                                        const void *lElement,
                                        const short lDataLen)
{
   Rect                 sicnBox;
   MyCustomDataRecPtr   cellData = (MyCustomDataRecPtr) lElement;

   sicnBox.left = lRect->left + 3;
   sicnBox.top  = lRect->top - 1;
   sicnBox.right  = sicnBox.left + 16;
   sicnBox.bottom = sicnBox.top + 16;

   Handle h = ::GetResource('SICN', cellData->iconID);
   if (h != NULL) {
      char saveState = ::HGetState(h);
      ::HLock(h);

      BitMap srcBits = { *h, 2,              // baseAddr, rowBytes
                         {0, 0, 16, 16} };   // bounds
      GrafPtr  port;
      ::GetPort(&port);
      ::CopyBits(&srcBits, &(*port).portBits, &srcBits.bounds, &sicnBox,
                 srcCopy, NULL);

      ::HSetState(h, saveState);
   }
   ::MoveTo(lRect->left + 24, lRect->top + 10);
   ::DrawString(cellData->name);
}
```

for instance, it can't accommodate script systems like Hebrew and Arabic by displaying its twist-down buttons on the right instead of the left — but it's essentially similar. The important implementation difference is that a sublist doesn't have to be present in memory before it's displayed: the contents are fetched when the sublist is expanded.

Figure 5 shows the basic data structure representing a twist-down list. Each cell has an indentation level and a flag byte, followed by a variable-length field holding the cell's data. The kHasSubList flag in the flag byte tells whether the cell has a sublist associated with it; if so, the kIsOpened flag indicates whether the sublist is currently open (expanded) or closed (collapsed). Cells with a sublist will be drawn with a triangular twist-down button pointing to the right if the sublist is currently closed, or down if it's open.

To expand or collapse a cell's sublist when the user clicks the triangular button, we override the list's ClickSelf method (inherited from the built-in PowerPlant class LListBox). Expanding a cell adds new cells to the list following it, with an indentation level that's 1 greater than its own. Collapsing a cell scans forward through the list and removes all immediately succeeding cells with higher indentation levels. The detailed code is too involved to show here, but if you're interested, you can find it on the CD in the file CTwistDownListBox.cp.

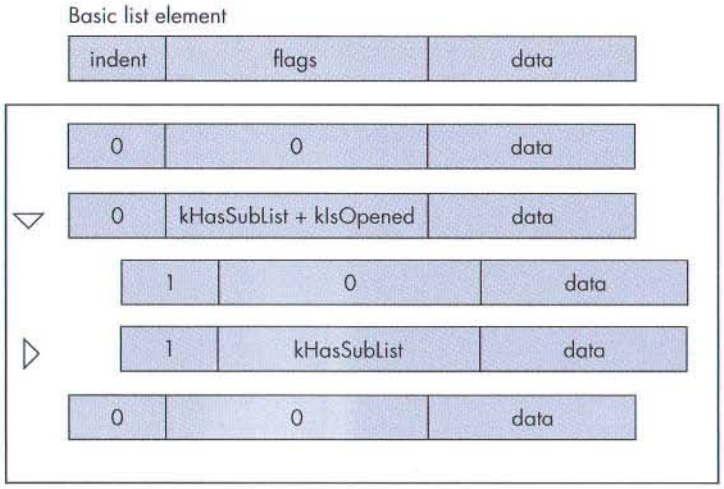


Figure 5. Structure of a hierarchical list

Listing 4 shows the redefined version of the DrawElementSelf method, inherited from CCustomListBox. First we check the cell's flags to see if it has a sublist; if so, we draw the triangular button in the appropriate form, depending on whether the sublist is open, closed, or in transition. The actual drawing of the cell's contents is factored out into a separate method, DrawTwistedElement: this allows subclasses to override just the drawing routine itself, without having to duplicate the logic for drawing the triangle as well.

A SIMPLE EXAMPLE

The class CMyHierListBox is a subclass of CTwistDownListBox, strictly for demonstration purposes. It isn't a particularly realistic example, but it does show how to specialize CTwistDownListBox to implement a simple hierarchical list. Each level of the hierarchy just consists of the words "One" through "Five" (see Figure 6).

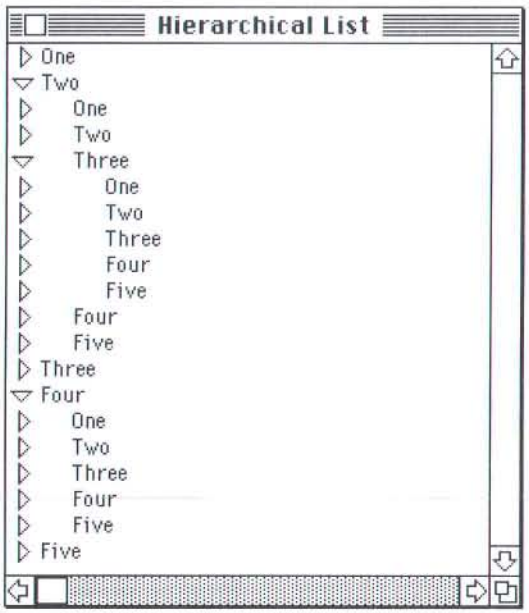


Figure 6. A hierarchical list

Listing 4. Drawing method for CTwistDownListBox

```
void CTwistDownListBox::DrawElementSelf (const Rect *lRect,
                                         const void *lElement,
                                         const short lDataLen)
// Draw a single list cell on the screen.
// Checks flags and draws triangular button if needed;
// calls DrawTwistedElement to draw cell contents.
{
   TwistDownRecPtr twistElement = (TwistDownRecPtr) lElement;

   if (TestTDFlag(twistElement->flags, kHasSubList)) {
      PolyHandle aPoly = NULL;
      aPoly = TestTDFlag(twistElement->flags, kIsOpened) ?
                  sOpenedPoly : sClosedPoly;
      if (TestTDFlag(twistElement->flags, kDrawIntermediate))
         aPoly = sIntermediatePoly;
      if (aPoly)
         DrawTriangle(aPoly, TestTDFlag(twistElement->flags,
                     kDrawFilled), lRect->top + 1,
                     lRect->left + kTriangleOutsideGap);
   }

   // Adjust pen position for triangle and indent.
   ::MoveTo(lRect->left + triangleWidth + 2 +
            twistElement->indent * kIndentOffset, lRect->top + 10);
   DrawTwistedElement(lRect, twistElement, lDataLen);
}

void CTwistDownListBox::DrawTwistedElement (const Rect *lRect,
                                    const TwistDownRecPtr twistElement,
                                    const short lDataLen)
// Draw contents of a single list element.
// Default version just draws text; override for other types of data.
{
   ::DrawText(twistElement->data, 0, lDataLen - TwistDownRecSize);
}
```

Every cell automatically has a sublist; you can keep opening sublists as long as you like, to unlimited depth (or until you run out of memory, anyway!).

The only method CMyHierListBox needs to redefine is ExpandElement (see Listing 5). The new version simply adds five new rows of dummy data following the cell being expanded. We don't have to override any other methods, since the superclass, CTwistDownListBox, already implements text elements by default. (For simplicity and clarity, we've simply hard-coded the words "One" through "Five" directly into the program itself; in real life, we would want to define them as resources to make modification and localization easier.)

A MORE INTERESTING EXAMPLE

This example is borrowed directly from Martin Minow's article. CMyDiskListBox is a subclass of CTwistDownListBox that displays the folder and file hierarchy on all currently mounted disk volumes, with each line preceded by a small icon as in our earlier CMyCustomListBox example (see Figure 7).

Listing 5. Cell expansion method for CMyHierListBox

```
static StringPtr myElements[] = {
                    "\pOne", "\pTwo", "\pThree", "\pFour", "\pFive"
                };

void CMyHierListBox::ExpandElement (const Cell theCell)
{
   short            num = sizeof(myElements) / sizeof(StringPtr),
                    i,
                    indent = 0;
   Cell             cell = {0, 0};
   Byte             buffer[100];
   TwistDownRecPtr  thisTwist = (TwistDownRecPtr) GetCellPtr(theCell);
   TwistDownRecPtr  anElement = (TwistDownRecPtr) buffer;

   if (thisTwist)
      indent = thisTwist->indent + 1;

   ::LAddRow(num, theCell.v + 1, mMacListH);

   for (cell.v = theCell.v + 1, i = 0; i < num; i++, cell.v++) {
      anElement->indent = indent;
      anElement->flags = 0x01;   // has sublist
      ::memcpy(anElement->data, myElements[i] + 1, *myElements[i]);
      ::LSetCell(anElement, sizeof(TwistDownRec) - 2 + *myElements[i],
                 cell, mMacListH);
   }
}
```

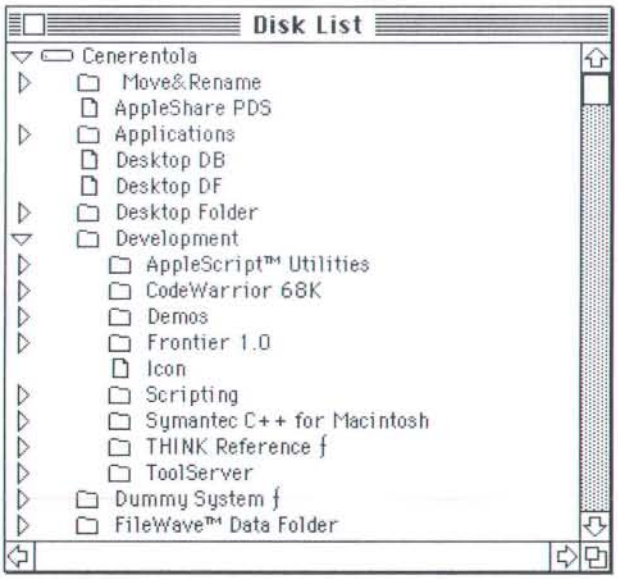


Figure 7. A disk list

The following structure contains the data for each cell of the list:

```
typedef struct {
   TwistDownHeader   hd;
   long              refNum;
   char              vRefNum;
   Byte              tag;
   char              name[2];
} DiskListRec, *DiskListPtr;
```

The data structure in the first field, TwistDownHeader, is inherited from the superclass (CTwistDownListBox) and contains the indentation level and the flag byte. Next come the file and volume reference numbers. The tag byte identifies this item as either a file, a folder, or a volume. The name field actually has variable length: when drawing the cell's contents, we know the total length of the cell data, so we can deduce the true length of the name.

> **Space is at a premium** when dealing with the List Manager, because of its 32K limit on the total size of a list record and its associated cell data. That's why we've saved a little space in the definition of the DiskListRec structure by making the vRefNum field a character instead of a short integer. This sacrifices a bit of speed when retrieving or storing the volume reference number, but avoids wasting an extra byte for word alignment. For the same reason, we also specify 68000 alignment in our Power Macintosh implementation. •

Because CMyDiskListBox draws more than just text in each cell of the list, it must override the DrawTwistedElement method inherited from CTwistDownListBox. Listing 6 shows the new version, which reads in the appropriate small icon from the system resource file, calls a subsidiary function, PlotSICN, to draw it, and then draws the text to go with it.

We also need to override the ExpandElement operation to look up the contents of a folder, using the file system calls PBHGetVInfo and PBGetCatInfo, and insert them in the list. (Our constructor method calls this same function to initialize a newly created list to the set of currently mounted volumes.) You'll find the code for this operation in the file CMyDiskListBox.cp on the CD. The new ExpandElement method doesn't allocate memory or do anything else that needs to be cleaned up later, so there's no need to override its companion method, CollapseElement.

There's a change in the resource file, too: instead of just using a class alias, as we did for our other example classes, we define our own resource template for CMyDiskListBox.

```
case DiskListBox:
   key literal longint = 'dlst';
   PP_ListBoxData;
```

This definition is placed in a separate *template file*, which must be defined in the tool server script and exported, so that it will be included with PowerPlant's own templates. We can then refer to our template by name when defining the 'PPob' resource in our main resource description (.r) file:

```
ObjectData {DiskListBox {
   ...
}}
```

Listing 6. Drawing method for CMyDiskListBox

```
const short sicnID[] = {      // system icon IDs
   -3995,   // tag_disk
   -3999,   // tag_folder
   -4000    // tag_file
};

const Size DiskListRecSize = sizeof(DiskListRec) - 2;
                                          // don't count the name field

void CMyDiskListBox::DrawTwistedElement (const Rect *lRect,
                                         const TwistDownRecPtr lElement,
                                         const short lDataLen)
// Draw contents of a single list element, including icon.
{
   Point pen;

   ::GetPen(&pen);

   Handle h = ::GetResource('SICN', sicnID[DiskListPtr(lElement)->tag]);
   if (h != NULL) {
      Rect box = {lRect->top - 2, pen.h, lRect->top + 16 - 2,
                  pen.h + 16};
      ::PlotSICN(&box, h);
   }
   ::Move(21, 0);
   ::DrawText(DisListPtr(lElement)->name, 0,
              lDataLen - DiskListRecSize);
}

static void PlotSICN (Rect *rect, Handle sicnList)
// Draw the icon for a list element.
{
   GrafPtr  port;
   char saveState = ::HGetState(sicnList);
   ::HLock(sicnList);

   BitMap srcBits = { *sicnList, 2,          // baseAddr, rowBytes
                      {0, 0, 16, 16} };      // bounds
   ::GetPort(&port);
   ::CopyBits(&srcBits, &(*port).portBits, &srcBits.bounds, rect,
              srcCopy, NULL);

   ::HSetState(sicnList, saveState);
}
```

In this case, defining our own template yields the same results as using a class alias, so it's just a matter of taste. But if you want flexibility, defining your own templates is the way to go: you can add or change existing resource definitions to suit your own classes. Just look at the various LPane subclass implementations in PowerPlant to see how easy it is!

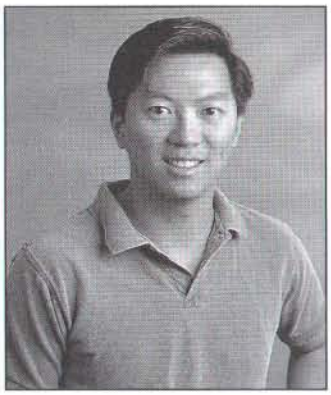
YOU TAKE IT FROM HERE

I hope you can see by now that the object-oriented approach makes it easy to define new kinds of hierarchical lists for your applications. The examples in this article are just a starting point: the rest is up to you.

If you're a true object aficionado, you'll want to make your list elements full-fledged objects instead of just simple data structures. You could modify our disk list example to display its icons in color instead of black and white, or to use each application or document's actual icon instead of the generic ones from the system resource file. Or how about letting the user drag and drop files from one folder to another within the list box? (PowerPlant provides predefined classes to support drag and drop, so building it into your application is easier than you might think. I know, because I've done it.)

The possibilities are limited only by your imagination. So get to work and see what *you* can dream up!

Thanks to our technical reviewers Nitin Ganatra, Martin Minow, Avi Rappaport, and Jeroen Schalk.•



SOMEWHERE IN QUICKTIME

Choosing the Right Codec

JOHN WANG

As described in Chapter 3 of *Inside Macintosh: QuickTime*, the Image Compression Manager performs compression and decompression by invoking image compressor and decompressor components. These components, called *codecs*, present a standard interface to the Image Compression Manager using the Component Manager. But each codec is unique because each implements a different compression and decompression algorithm. Codecs can vary greatly in the three major characteristics that are used to judge image compression algorithms: compression ratio, compression speed, and image quality. QuickTime 2.0 ships with eight codecs that can be selected for use by your QuickTime application under various conditions. In addition, users or third-party products can install custom codecs into the system by simply placing them in the Extensions folder.

With all the choices for image compression and decompression, the only way to choose the best codec for a particular purpose is to have some understanding of all the codecs available on your system. *Inside Macintosh* provides general descriptions of the standard QuickTime codecs, but detailed information is important in making an intelligent codec selection. The way to find this detailed information is to communicate programmatically with the codec and request its capabilities through the codec call CDGetCodecInfo. As described in *Inside Macintosh: QuickTime Components*, this call returns a compressor information structure.

For example, information about what pixel depths a compressor supports for storing image data is important to choosing the right codec. If your application creates and compresses a picture, the decision of whether to create the picture in 8-bit, 16-bit, or 32-bit color can be based partially on what pixel depths the compressor supports. If the compressed data can store only 16 bits of color information, it would be inefficient to create a picture with 32 bits of color.

Accompanying this column on this issue's CD is the sample application GetCodecInfoApp, which (by calling CDGetCodecInfo) allows you to easily obtain detailed information about codecs installed in your system. I'll discuss GetCodecInfoApp and point out some characteristics that should be considered in choosing a codec.

USING CODECS

There are actually two separate parts to a codec: one for the compression and one for the decompression. Not all codecs provide both; nevertheless, all compressor and decompressor combinations are referred to as codecs. One example of a decompression-only codec is the codec that comes with the QuickTake 100 digital camera from Apple. The hardware in the QuickTake 100 camera performs the compression and downloads compressed data to the Macintosh. The Macintosh only needs to perform decompression.

A compressor is a component of type compressorComponentType ('imco') and a decompressor is a component of type decompressorComponentType ('imdc'). Detailed information on writing a codec is provided in Chapter 4 of *Inside Macintosh: QuickTime Components*. But to select the appropriate codec to use, you don't need to do any programming; you can simply use GetCodecInfoApp, without any need to understand how it was written. This application creates a text file containing a report of all the codec components installed in your system. For example, the output for the Cinepak codec looks like this:

```
Compressor Name: Cinepak
-----------------------------------
- version = 1
- revisionLevel = 1
- vendor = appl
- compressionAccuracy = 128
- compressionLevel = 128
- minimum height = 1
- minimum width = 1
- compress pipeline latency = 0
```

JOHN WANG (AppleLink WANG.JY) used to be a proud member of the PIGs (Printing, Imaging, and Graphics group) in Apple's Developer Technical Support group. But he decided that there are other challenges in life and programming. So now John spends his entire day waiting for MPW to compile code that he's writing in his software engineering role in the Image Capture group. Just in case you fail to notice, we're sure he'd like us to point out that he makes a gratuitous plug for his group's product, the QuickTake 100 digital camera, in this column.•

```
- compression capabilities:
    directly compresses 32-bit pixel maps
    supports temporal compression
    can recompress images without accumulating
        errors
    can rate constrain to caller defined limit
- compression format:
    can store images in 24-bit color
    can store images in 8-bit grayscale
    can store custom color table
    compressed data requires non-key frames to be
        decompressed in same order as compressed
- estimated compression speed:
    640x480 32-bit RGB = 11485 milliseconds

Decompressor Name: Cinepak
-----------------------------------
- version = 1
- revisionLevel = 1
- vendor = appl
- decompressionAccuracy = 128
- minimum height = 1
- minimum width = 1
- decompress pipeline latency = 0
- decompression capabilities:
    directly decompresses into 32-bit pixel maps
    supports temporal compression
    can recompress images without accumulating
        errors
    can rate constrain to caller defined limit
- decompression format:
    can decompress images from 24-bit color
        compressed format
    can decompress images from 8-bit grayscale
        compressed format
    can store custom color table
    compressed data requires non-key frames to be
        decompressed in same order as compressed
- estimated decompression speed:
    640x480 32-bit RGB = 56 milliseconds
```

GetCodecInfoApp gets information about codecs by calling the codec's CDGetCodecInfo function, which all codecs must support; if you're writing a codec, it's important to report your capabilities with this function. To measure the codec's speed, the application actually passes it an image to compress or decompress, and reports the result.

The Image Compression Manager function GetCodecInfo can also be used to obtain information about codecs, but only for compressor codecs; you won't be able to get information about decompression-only codecs with GetCodecInfo.•

An example of a characteristic you can determine with GetCodecInfoApp is what pixel depths the decompressor can decompress directly into. This is important because it affects the speed of the image decompression. If the codec can't decompress directly into the destination pixel map, the Image Compression Manager will have to decompress into an offscreen buffer and move the image data into the destination after converting the pixel depth. This results in additional memory and processor bandwidth requirements. If you know exactly what pixel depths a decompressor supports, you can set up the destination for the best performance.

Most codecs support only a limited number of pixel depths for the compressed data storage format. For example, the Video Compressor will store image data only in 16-bit color. If you compress a 32-bit color image, you'll lose information, since the compressed format will store the equivalent of 16 bits of data. The pixel depth for the compressed data storage format also determines which of the different compression settings are available — for example, the pixel depth pop-up menu for Compression Settings displayed by the standard image-compression dialog component (used, for example, by Picture Compressor, an application that's part of the QuickTime Starter Kit) will only allow you to choose Color for the Video Compressor. The Animation Compressor is one of the few compressors that will store compressed data in nearly all pixel depth formats: Black and White, 4 Grays, 4 Colors, 16 Grays, 16 Colors, and so on.

When compressing movies, you'll often want to select a codec that supports temporal compression; not all codecs do. *Temporal compression* is the use of frame differencing to compress consecutive image frames by skipping data that doesn't change from frame to frame. Temporal compression is useful only for sequences of images stored as QuickTime movies. Knowing which codecs support temporal compression will allow you to choose the best codec for compressing sequences.

If you're compressing pictures with scientific data, it may be extremely important that there be no image quality loss. In this case, you'll want to look for a codec that supports *lossless* compression. For example, the Photo Compressor (JPEG codec) is a lossy codec because even at the highest quality setting, there may still be some loss of image quality. On the other hand, the Animation Compressor is lossless at higher quality settings and will preserve every pixel value.

There are many additional features a codec may support that are important to know. For example, certain codecs will support data spooling so that only portions of the compressed data need to be read into memory at any one time. This can be a requirement

when working with very large compressed images that will be displayed in systems with limited memory. Another example is support for stretching to double size during decompression. This is extremely useful, since the performance is much greater if the scaling is performed during decompression rather than as a separate step after decompression.

SOME RECOMMENDATIONS

For most video clips, the Cinepak Compressor is the recommended codec. As you can see from GetCodecInfoApp's report, this codec is very slow in compression. However, its decompression speed and compression level are excellent, making it the best choice for most video data for CD-ROM playback.

An alternative to Cinepak is the Video Compressor. Since its compression speed is fairly quick, it's better for an application that requires fast compression.

If your source material is animation graphics in a movie, there are several compressors that may do the job. The Animation Compressor and Graphics Compressor may be equally suitable. In this case, you may need to experiment to determine which is the best codec to use.

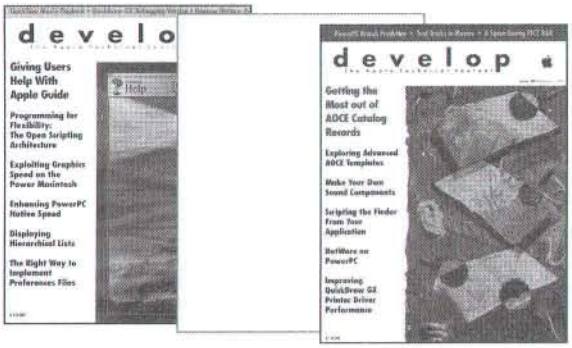
Finally, if you're compressing photo images, the Photo Compressor is the best codec to use. It has only moderate compression and decompression speed, but the compression ratio and quality are excellent and the compression ratio scales accordingly with image quality. If you want better image quality at the expense of larger compressed data size, you can easily achieve this with the Photo Compressor.

PRESSING ON

If you're writing a codec, you can see from this column that it's very important to properly report the codec's capabilities; GetCodecInfoApp may be useful for you to verify that your codec is doing this properly. For the rest of you, I hope this column has provided some insight on how to choose the right codec for producing the best movies and compressed images.

Thanks to Peter Hoddie, Don Johnson, Kent Sandvik, and Nick Thompson for reviewing this column.•

Missing something?



Are there issues of *develop* that have passed you by? If you'd like to complete your *develop* collection, full-color, bound copies are available. (Back issues are also on the *develop Bookmark* CD and the Reference Library edition of the *Developer CD Series*.) To order printed back issues, send $13 per issue in the U.S. (or $20 outside the U.S.) to *develop* Back Issues, P.O. Box 531, Mount Morris, IL 61054-7858. Or call 1-800-877-5548 in the U.S. or (815)734-1116 elsewhere. *Supplies are limited. Please allow 4 to 6 weeks for delivery.*

Issue 1 Color; Palette Manager; Offscreen Worlds; PostScript; System 7; Debugging Declaration ROMs

Issue 2 C++ Objects; Object Pascal; Memory Manager; MacApp; How to Design an Object-Based Application; C++ Style Guide; The GS/OS Cache

Issue 3 ISO 9660 and High Sierra; A Mixed-Partition CD; Accessing CD Audio Tracks; Comm Toolbox; Macintosh Display Card 8•24 GC; PrGeneral

Issue 4 Device Driver in C++; Polymorphism in C++; A/ROSE; PostScript; Apple IIGS Printer Driver

Issue 5 (Volume 2, Issue 1) Asynchronous Background Networking; Palette Manager; Macintosh Common Lisp

Issue 6 Threads; CopyBits; MacTCP Cookbook: Constructing Network-Aware Applications

Issue 7 QuickTime 1.0; TrueType; Threads and Futures; C++ Objects in a World of Exceptions

Issue 8 Curves in QuickDraw; Date and Time Entry in MacApp; Debugging; Hybrid Applications for A/UX

Issue 9 Color on 1-Bit Devices; The TextBox You've Always Wanted; Making Your Macintosh Sound Like an Echo Box; Simple Text Windows via the Terminal Manager; Tracks: A New Tool for Debugging Drivers

Issue 10 Apple Event Objects; PostScript Enhancements for the LaserWriter Font Utility; Drawing in GWorlds; The Optimal Palette

Issue 11 Asynchronous Sound; Multibuffering Sounds; Exceptions; NetWork: Distributed Computing

Issue 12 Components; Time Bases; Apple Event Coding Through Objects; Globals in Standalone Code

Issue 13 Asynchronous Routines; QuickTime and Component-Based Managers; Macintosh Debugging Revisited; Adventures in Color Printing; DeviceLoop

Issue 14 Writing Localizable Applications; 3-D Rotation Using a 2-D Input Device; Video Digitizing Under QuickTime; Making Better QuickTime Movies

Issue 15 QuickDraw GX (Getting Started; Printing Extensions; PostScript); Component Registration; Floating Windows; Working in the Third Dimension

Issue 16 Making the Leap to PowerPC; PowerTalk; Drag and Drop From the Finder; Color Matching With QuickDraw GX; International Number Formatting

Issue 17 Proto Templates on the Newton; Standalone Code on PowerPC; Debugging on PowerPC; Thread Manager; Window Zooming

Issue 18 Apple Guide; Open Scripting Architecture; Graphics Speed on the Power Macintosh; Displaying Hierarchical Lists; Preferences Files

Issue 19 OpenDoc Part Handlers; PowerPC Memory Usage; Designing for the Power Macintosh; Adding QuickDraw GX Printing to QuickDraw Applications; QuickDraw GX Bitmaps; Inheritance in Scripts

Issue 20 Getting the Most out of AOCE Catalog Records; Exploring Advanced AOCE Templates; Make Your Own Sound Components; Scripting the Finder From Your Application; NetWare on PowerPC

Macintosh Q & A

Q *I'm about to write my first QuickDraw GX printer driver (for a laser printer) and plan to base it on the sample code "Generic LaserWriter." Is there anything I need to know about that code?*

A There are two bugs that you might need to know about, depending on which version of the sample you're using. Both are very easy to fix.

- In versions previous to those in the QuickDraw GX Developer's Kit (which is now part of the Mac OS Software Developer's Kit), a device ID of $88 is specified in the file ChooserSupport.a. For QuickDraw GX, the device ID should always be $A9. (This bug also slipped through in the "LaserWriter--custom dialogs" sample in the Developer's Kit.)
- For all the versions of the code in the Developer's Kit, we improved the file ChooserSupport.c by adding better error handling, making everything universal, and removing unnecessary code; however, we forgot to add ALRT -4095 and DITL -4095 to the driver. If the Chooser code fails, it tries to put up a "You're hosed" alert, which doesn't exist. You should still use this improved version if you can; you can just copy the missing resources from any of the other sample LaserWriter drivers in the Developer's Kit.

Q *I've just ported my application to the Power Macintosh and I want it to optionally use the Speech Manager. But if I try to launch the application when the Speech Manager isn't installed, the Finder puts up a dialog that says "The application 'GonzoApp' could not be opened because 'SpeechLib' could not be found." How can I get my application to launch in this case?*

A The Finder refuses to launch your application because the Code Fragment Manager (CFM) believes your application is "hard linked" to SpeechLib. The CFM assumes your application can't run at all without SpeechLib and that all references to SpeechLib must be resolved. Since it can't locate SpeechLib, it refuses to launch your application.

To get around this, the CFM supports a concept called "weak linking." In this case, it allows references to the library to be unresolved at run time, and it's up to the application (that is, your code) to check that it's safe to make calls to the library. If you don't check, your application will crash the first time it tries to call the library.

The best way to check that a weak-linked library is installed is to check that the address of a function in the library has been resolved. For example:

```
Boolean HasSpeechLib()
{
   /*
    * Check the address of some function in the library.
    * SpeechManagerVersion is convenient.
    * kUnresolvedSymbolAddress is defined in FragLoad.h.
    */
   return (SpeechManagerVersion != kUnresolvedSymbolAddress);
}
```

Note that this is not a replacement for using Gestalt to determine whether services are available; you should still use Gestalt to determine whether the Speech Manager is installed. The above code is an additional check for native applications, to ensure that the necessary library functions are available.

How you weak-link your application to a particular library depends on your development environment. Using the PPCC tools, you would specify weak linking with the following option to MakePEF:

```
-l SpeechLib.xcoff=SpeechLib~
```

Note the tilde (~) character at the end: this specifies a weak-linked library. In Metrowerks CodeWarrior, when you add SpeechLib to the project, the little pop-up menu on the right side of the project window will have an "Import weak" option that you can use to enable weak linking. Other development environments may use different methods for designating a weak-linked library.

Q *To make my application localizable, I want to use ExtendedToString and StringToExtended to convert floats to strings and strings to floats. These routines, though, use the SANE extended format, which is quite archaic. What's the best way to convert a float to an extended to pass to ExtendedToString? It should compile on both 680x0 and PowerPC machines with MPW, Metrowerks, and THINK C.*

A On PowerPC machines, **extended80** and **extended96** do not exist, except as structures. There are a number of (PowerPC-only) conversion routines in fp.h, like **x80told**. Your formatting code can be written as follows:

```
#include <fp.h>
#include <TextUtils.h>

void LongDoubleToString (long double ld,
                         const NumFormatString *myCanonical,
                         const NumberParts *partsTable,
                         Str255 outString)
{
#if defined(powerc) || defined (__powerc)
   extended80  x;

   ldtox80(&ld, &x);
   ExtendedToString(&x, myCanonical, partsTable, outString);
#else
   #ifdef mc68881
      extended80  x;

      x96tox80(&ld, &x);
      ExtendedToString(&x, myCanonical, partsTable, outString);
   #else
      ExtendedToString(&ld, myCanonical, partsTable, outString);
   #endif
#endif
}
```

Note that **long double** is equivalent to **extended80** or **extended96** on 680x0, and 128-bit **double-double** on PowerPC. If you want to format a double, just pass it in and the compiler will take care of converting double to long double or double to extended.

SANE.h is being replaced by fp.h, for both 680x0 and PowerPC. This issue's CD contains the libraries and interfaces for this new numerics package, which

implements the Floating-Point C Extensions (FPCE) proposed technical draft of the Numerical C Extensions Group's requirements (NCEG/X3J11.1).

For more information on how to convert a SANE program to PowerPC, see *Inside Macintosh: PowerPC Numerics*, Appendix A. In principle, you should replace all use of **extended** with **double_t**. The **double_t** type maps back to **long double** for 680x0, which is the same as **extended**; for PowerPC, it maps to 64-bit **double**, which is the fastest floating-point format. Only when you read or write 80- or 96-bit SANE numbers to files, or when you want to use any of the Toolbox routines that take an 80-bit extended parameter, do you need to use conversion routines.

Q *I'd like to automatically configure printing in portrait or landscape mode without requiring the user to go through the Page Setup dialog. How do I go about doing this?*

A The traditional Macintosh printing architecture provides no support for changing the page orientation programmatically. At the time the Printing Manager was designed (about ten years ago!), there was kind of an overreaction regarding the principle of the user being in control, so we've had to live without a dependable way of changing a print record without the user's help. The good news is the QuickDraw GX printing architecture does support changing print dialogs programmatically, and QuickDraw GX will eventually be replacing the old printing architecture.

If you're interested in finding a workaround in the traditional printing environment, the only one we can offer is to prepare a couple of print records with the desired settings ahead of time and include them in your application. Then you can select the appropriate one based on the high-order byte of the TPrStl.wDev field of a validated print record. If the current printer driver has a value you don't know about, ask the user to go through the Page Setup dialog once to set landscape mode. Your application can then save the print record after it's filled out by PrStlDialog (again, indexed by the high byte of the wDev field).

The best method for saving the print record is to save it as a resource in your document's resource fork. Make your resource type something different from those used by the Printing Manager, to prevent the Printing Manager from getting confused and grabbing the wrong resource.

Remember, you need to be careful: watch the high byte of the wDev field, and call PrValidate before passing a print record to PrOpenDoc. Also, take a look at *Inside Macintosh: Imaging With QuickDraw*; pages 9-17 and 9-18 provide some valuable information.

Q *I want to create a mask for a picture, such that the mask is 0 wherever the picture's pixels are pure white, and 1 everywhere else. My first try was to simply use CopyBits to copy the rectangle enclosing the PICT onto a same-sized rect in a 1-bit-deep offscreen world. This didn't work, however, as the yellows get transformed to 0, which is not what I want. I tried various transfer modes with CopyBits (from 0 to 100) to no avail. The SeedCFill and the CalcCMask routines don't seem to be what I want either, because it appears that their masks have to be keyed off a certain point or side(s). I can take the brute force approach and go through the pixels of the PICT one by one, checking to see if they're white and setting the mask accordingly, but this seems insane. Is there a good method for doing this?*

A The way to do this is to install a custom color search procedure, then call CopyBits to copy into the 1-bit GWorld and let the search proc determine the color to use. The search proc would simply look at the color asked for and return white if it's white or black if it's nonwhite. See "Color Manager" in *Inside Macintosh: Advanced Color Imaging* (on this issue's CD and forthcoming in print from Addison-Wesley).

Q *I heard that you can use LaserWriter 8.1.1 with QuickDraw GX with some patch. Is this true? If so, how?*

A You can install an option called QuickDraw GX Helper. To do this, launch the installer document for QuickDraw GX and choose Custom Install from the pop-up menu in the top left of the window. A list appears with a bunch of options, with checkboxes next to them. Click the small triangle next to the QuickDraw GX Utilities option; then check the QuickDraw GX Helper option below that.

After you install and reboot, Helper will be available to you. It works only with old-style "classic" printing applications (though there's nothing classic about the old Printing Manager :-); if you're using a QuickDraw GX–savvy application, Helper won't help. When you're in a "classic" printing application, there's a new option, Turn Desktop Printing On (or Off) in the Apple menu.

Be aware that this is more than a patch. Literally, it's an unpatch! It removes some of the QuickDraw GX patches and is really a bit of a hack, provided for convenience. Because of this, there isn't much of an interface. For instance, when desktop printing is turned off, the printer used is the one selected the last time that "classic" driver was used. To change to a different printer, you need to reboot without QuickDraw GX, choose a new printer in the Chooser, then reboot again with QuickDraw GX.

Here are some additional things to note:

- Helper will choose LaserWriter 7.x over LaserWriter 8 (because LaserWriter 7.x was called "LaserWriter," whereas LaserWriter 8.x is called "LaserWriter 8," and the former is ordinally before the latter). As you probably know, QuickDraw GX can't tolerate multiple printer drivers with the same creator codes, so it just picks the first one. Therefore, you should ensure that you have only one driver of each type in your Extensions folder.
- Of course, before turning off desktop printing you need to ensure that you have the appropriate desktop printer selected: if you want LaserWriter 8.1.1 selected when you turn off desktop printing, make sure a LaserWriter is the selected desktop printer; if you want ImageWriter to be selected when you turn off desktop printing, make sure an ImageWriter is the selected desktop printer; and so on.
- Helper works only with Apple drivers. If you need to print with another driver, rebooting without QuickDraw GX is the only option.

Q *I'm trying to get a gxFont ID based on a given font name (for example, "Times"), and I've run across a confusing situation using GXFindFonts. Below is the call I'm using, and it gives an "inconsistent parameters" error. Can you tell me why this error occurs and what I'm doing wrong? Using gxFullFontName doesn't work in finding the font in this case. Using gxNoFontName gives errors with or without the font family ID; when should this indicator be used?*

```
GXFindFonts(theFontFamilyID,        // font family
            gxNoFontName,           // font name
            gxMacintoshPlatform,    // font platform
            1,                      // default -> gxRomanScript
            1,                      // default -> gxEnglishLanguage
            strlen(name),           // name length
            (unsigned char*)name,   // the name
            1,                      // matching font
            1,                      // number of matches
            &fontID                 // font reference);
```

A GXFindFonts is the Swiss Army knife of font-finding routines. You can use it to find one font or a set of fonts. You can have it base the search on nearly any combination of family name, style name, script type, or other font properties. Unfortunately, the combination of arguments it expects in order to work in its myriad ways can be a bit confusing — it's easy to cut your finger on one of the blades and get an "inconsistent parameters" error.

The first argument passed (font family) is itself a gxFont structure. This argument, if not nil, allows you to restrict the search to those fonts that have the same family as the argument.

Each gxFont has several types of name string associated with it. As documented on page 7-7 of *Inside Macintosh: QuickDraw GX Typography*, there's a font family string, style string, unique name string, full font name string, and so on for each font. The second argument passed to GXFindFonts (the meaning argument) is one of the gxFontNames constants listed on page 7-79. This argument, if not gxNoFontName, specifies which of the font's names to compare with the seventh argument to determine a match. If the second argument is gxNoFontName, GXFindFonts ignores the seventh argument and assumes that you're basing your search on other criteria, such as platform, script, or language (arguments 3, 4, and 5, respectively).

The seventh argument (name), if not nil, is the name string that you want to search for, and the sixth argument (nameLength) is the length of that string.

GXFindFonts may find more than one font that matches the criteria. The eighth argument (index) is the index of the first of these fonts that you want GXFindFonts to return. The ninth argument (count) is the maximum number of fonts that you want GXFindFonts to return. So, for example, if you specify criteria that match ten different fonts, but you want GXFindFonts to return only the fifth, sixth, and seventh of these fonts, you'd pass 5 for the index and 3 for the count.

The tenth argument (fonts) is a pointer to a buffer in which GXFindFonts will return the matches it finds. This argument can be nil if you don't want the fonts returned — for example, if you just want to find out the number of matches to your desired search (the number of matches is returned as the function result).

You got an error because you passed in a string to search for (in the sixth and seventh arguments) and yet specified gxNoFontName in the second argument. These arguments are inconsistent; fortunately you were using the debugging version of GX and received an error.

But, to return to my pocket knife analogy, sometimes it's simpler and safer to use a Buck knife instead of a Swiss Army knife (only one blade, and it locks!). If

getting a font ID based on a font name is all you need to do, you should consider using the FindCNameFont function in font library.c. With this routine you can simply call

```
fontID = FindCNameFont(gxFullFontName, name);
```

instead of the nasty, multiple-parameter GXFindFonts. There are several other useful tools in font library.c which are also worth a look.

Q *Perhaps you can clear up a long-running dispute we've had in our office. Long ago I read that the "Mac" in MacsBug doesn't stand for Macintosh, but is an acronym for something that starts with "Motorola." Please put my mind to rest.*

A MacsBug stands for **M**otorola **a**dvanced **c**omputer **s**ystems de**bug**ger.

Q *My application is going to support only the four "required" Apple events plus one custom event (the application will have an 'aete' resource). What's the desired behavior of the 'pdoc' event? The real question is, if the user selects a file type that this application created and chooses Print from the File menu, should a dialog box be presented to the user? Obviously, if the application receives the event from another application or a scripting system, we don't want to display the Print dialog (possibly on a remote machine). Should the behavior be different if the Finder sends the event? Should there be one or two ways of handling the event?*

A Here's how your application should behave upon receiving a 'pdoc' event: If all you get is a 'pdoc', you should just print without user interaction. The user, or whoever, is just using you as a service to print a document. Start your printing without any dialogs, using whatever default print record you get from PrDefault. You do not have to quit; the Finder will send you a 'quit' Apple event as the next event you get.

If you're already running (that is, you were started with an 'oapp' or 'odoc' event) and you get a 'pdoc' event, you should treat that one in the same way as a user Print menu selection. You might be required to put up a dialog. Always remember to call AEInteractWithUser before putting up a print dialog (or any dialog) to be sure you have the right to interact, and be prepared to use default actions if you can't interact.

Q *The Macintosh Quadra 630 Developer Note, page 68, says "You should be familiar with the ATA IDE specification, ANSI proposal X3T9.2/90-143, Revision 3.1." Where can I find this document?*

A Apple's implementation of IDE is documented in the Developer Notes for the Macintosh Quadra 630 and the PowerBook 150. The ANSI IDE standard has been renumbered as X3.221 Revision 4A, April 93. You can order it (and other ANSI documents) from Global Engineering Documents; call 1-800-854-7179 in the U.S., or (303)792-2181 elsewhere. The cost is $25 plus shipping.

Q *I'm trying to print a document with over 60 different fonts in it under QuickDraw GX, and I get an (unknown) error -27881. The document doesn't print. Is this a bug? This document will, however, print on non–QuickDraw GX systems to the LaserWriter 8.1.1 driver and presumably other drivers as well. We haven't been able to find an equivalent to the non–QuickDraw GX "unlimited downloadable fonts" setting under*

QuickDraw GX. Is there a workaround to this problem? I realize that it's somewhat of a ridiculous case, but people do actually do this.

A This is not a QuickDraw GX bug. It seems likely, given the error you're seeing, that this is a problem with one of the fonts you're using.

If you use the **gerror** dcmd included with the QuickDraw GX Developer's Kit (or plow through the file graphics errors.h), you'll see that the error is font_scaler_streaming_aborted. This tells you that the streaming was aborted for some reason; a common reason would be that the naming table in the font is bad. You should be able to determine the exact cause of this using the PSMirrorFile extension (which you can find in the Printing Extensions folder in the Developer's Kit). This extension will log to a file the dialog with a PostScript printer; it really helps during debugging.

What all this implies is that one of the fonts you're trying to use is bogus. You need to determine which one is causing your problem and remove it. You may be able to do this by successively dividing your document into halves until you find the section of the document that's causing the problem.

Q *I'd like to know how to do chroma keying in QuickTime. I'm under the impression that this is possible, but haven't been able to figure out how by digging through Inside Macintosh.*

A All you need to do is call SetVideoMediaGraphicsMode, setting graphicsMode to transparent and setting opColor to whatever color you want to be transparent.

```
Media          theMedia;
MediaHandler   theHandler;
RGBColor       theKeyColor;

...      // Set up key color and get the track.

theMedia = GetTrackMedia(theTrack);
theHandler = GetMediaHandler(theMedia);
MediaSetGraphicsMode(theHandler, transparent, &theKeyColor);
```

Note that since QuickTime currently uses QuickDraw to do the compositing, this approach can be rather slow.

Q *I'd like to add a volume control style slider to my dialogs. I don't really want to have to implement my own CDEF since this must have already been done by many others. Is there anywhere I can pick one up?*

A In the Sample Code folder on this issue's CD, as part of AppsToGo, there's a sample program called Kibitz that uses a slider CDEF. You can use that one as the basis for writing your own control. You'll have the source as well as the object code. The code should be adaptable to your needs; if you don't like the way the slider looks, you can easily change it using a resource editor (the resource type of slider parts is 'cicn').

Q *I'm writing a real-time video application and would like to open file data forks for reading/writing at interrupt time (in a deferred task). What's the best call to do this?*

A You can open files at interrupt time as long as you make the PBHOpen, PBHOpenDF, or PBHOpenRF call asynchronously. These calls are always safe at interrupt time; they'll get queued if the File Manager is busy with another call, letting the current request complete before processing your request. See the article "Asynchronous Routines on the Macintosh" in *develop* Issue 13 for complete information. The article explains when, why, and how you should use functions asynchronously on the Macintosh.

Q *I was relying on sample code from Inside Macintosh to spool shapes when printing under QuickDraw GX, but it seems to be causing my application to crash. The code I'm using is in Inside Macintosh: QuickDraw GX Environment and Utilities, on page 1-22, Listing 1-6. Is that code correct?*

A The code is correct in the context in which it's given, but shouldn't be used for printing. Calling GXDisposeShape from the spoolProc while printing is what causes your crash: the QuickDraw GX printing mechanism disposes of the shapes for you.

Q *I'm trying to incorporate the minimal support for QuickDraw GX printing in my application, and I've run into a problem. For a start, I'm using Simple Sample GX, the sample code Dave Hersey wrote for his article "Adding QuickDraw GX Printing to QuickDraw Applications" in* develop *Issue 19. I made necessary modifications to this code for the printing method I'm using: our printing is basically 90% text output, which is paginated on the fly based on the size of the print page that's returned by the printing code. Crashes occurred, however, and I finally narrowed them down to DrawText or DrawString calls with only one character, or only one character followed by a space. Is this a bug?*

A Yes, it's a bug in the glyph code. We expect to fix this in the 1.1 release of QuickDraw GX (which should be available by the time you read this) but here's an easy workaround for QuickDraw GX 1.0.1 and 1.0: convert any glyph shapes to path shapes. In your spoolProc, after you get the shape type (as in Simple Sample GX), do this:

```
if (theShapeType == gxGlyphType)
   GXSetShapeType(currentShape, (theShapeType = gxPathType));
```

This will convert any glyphs to paths, and will circumvent the problems in the glyph code. I've verified that this works using Simple Sample GX and also in your test case. Note also that you will lose any hinting information, so the text may appear slightly different.

Q *What's the data that's passed when the Drag Manager sends a catalog directory, of type flavorTypeDirectory? The documentation says it's a DSSpec, but it's too small. Is it a packedDSSpec?*

A The documentation is wrong. It's a packedDSSpec, as you thought.

Q *Our application needed a source of uniform random numbers to generate statistical distributions, and we used the built-in Random function for this. A number of our users need to know the algorithm of Random because statisticians (as any mathematician) need to produce a numerical audit trail to document their work. I looked at the assembly*

code of the Random function and couldn't recognize the method, although it looks similar to a linear-congruent generator. Could you tell me the source of the Random function? If you can cite a book, that would be great!

A The Random function in QuickDraw is based on the formula

```
randSeed := (randSeed * 16807) MOD 2147483647
```

It returns a signed 16-bit number, and updates the unsigned 32-bit low-memory global randSeed. The reference used when implementing this random number generator was Linus Schrage, "A More Portable FORTRAN Random Number Generator," *ACM Transactions on Mathematical Software* Vol. 5, No. 2, June 1979, pages 132–138.

The RandomX function in SANE uses the iteration formula

```
r = (7^5 * r) mod (2^31 - 1)
```

as documented on page 67 of the *Apple Numerics Manual*, Second Edition.

Q *I want the users of my application to be able to grow the window without changing the aspect ratio. Is there a way to call GrowWindow but somehow be able to monitor the mouse position and modify the size of the rectangle while it's being dragged?*

A The best approach is to write your own custom replacement for GrowWindow that does what you want (see the snippet called GridWindowGrow on this issue's CD for an example of a replacement GrowWindow routine). Another option, easier but not really what you're after, is to allow "free" dragging and then enforce the aspect ratio in your subsequent call to SizeWindow.

Q *We're having a problem with MPW C++. We build our application requiring a 68020 processor or better and an FPU. The problem is that the MPW C++ compiler seems to create a CODE segment called Static_Constructors. This segment contains FPU-specific code and causes our program to crash on launch (ID 16) for machines without an FPU. Looking through code, I notice that at launch __RTInit is called, which in turn calls __CPlusInit. __CPlusInit loads the Static_Constructors segment and executes it, before the main segment is ever called. Can we fix this? How?*

A This is a known C++ problem and is mentioned in the Macintosh Technical Note "MPW C++ Q&As" (PT 555) under "C++ static constructors and checking for an FPU." A workaround is mentioned in the note but not in much detail; a little more information follows here.

You need to rename __CPlusInit to something else, and write your own replacement that calls the real one only if the FPU checks are passed. You can rename __CPlusInit (from RunTime.o) by using the Lib tool with the "-rn" option. Write your own version like this:

```
extern "C" void __CPlusInit(void)
{
   // Do the gestalt on FPU.
   Renamed_CPlusInit();
}
```

The **extern "C"** is relevant, since you don't want a C++ mangled name to link against.

Q *Under WorldScript, the itlbValidStyles field of the 'itlb' resource governs what styles can be applied to text, depending on the language of the font. I understand the reasoning — underlining makes no sense for vertical scripts, extended text makes no sense for cursive scripts, and so on. However, we need to underline Kanji text. How should we do it?*

A Underlining as implemented in QuickDraw was based on assumptions appropriate for Roman text — specifically, that the underline should be just below the baseline. Unfortunately, the Asian scripts don't have the same definitions for baseline, ascent, and descent, and this creates an irreconcilable problem. Excluding the Roman characters and some punctuation, all the characters in the Kanji font descend below the QuickDraw baseline, so when QuickDraw tries to draw the regular underline it gets broken (in the same way it does with Roman descenders like g, j, and p — only more so). Because it looked so bad, underline was disabled for the two-byte script systems. QuickDraw GX is the real solution to this complicated problem.

Barring that, you should just draw your own underlines manually, using QuickDraw, somewhere near the descent line. Exactly where is a matter of style. Because of that, we recommend that you do plenty of user testing, and be sure to look at other applications that do the same thing (MacWrite, PageMaker, QuarkXPress, WordPerfect, TurboWriter, and so on).

Two notes: First, Roman text that uses a Kanji font needs to follow this same convention, so that the underlines are consistent. (There may still be a problem when different fonts on the same line are underlined — the lines won't necessarily match up.) Also, if the text's style is set to underline, PostScript will still draw the underline in the traditional location, even though it's not displayed on the screen! If you're printing to a PostScript printer, be sure the text's style isn't underline or you'll end up with two underlines. Good luck!

Q *Why does a quarter have ridges on the edge?*

A Several hundred years ago, certain enterprising souls would shave the edges off of coins. They would then spend the coins as usual and sell the shavings as bulk valuable metal. In an effort to combat this, governments began decorating the sides of coins so that it would be apparent if the currency had been tampered with. Any shaved coin could be refused by a merchant. The U.S. mint followed suit and put edges on all silver and gold coinage (dollar denominations, half dollar, quarter, and dime) to deter shavers. Although currency became silver-copper clad in 1965, thereby making the metal much less valuable, the decision was made to retain the edging for tradition's sake.

These answers are supplied by the technical gurus in Apple's Developer Support Center. Special thanks to Brian Bechtel, Mark Harlan, David Hayward, Dave Hersey, Larry Lai, Martin Minow, Dave Radcliffe, Jeroen Schalk, and Nick Thompson for the material in this Q & A column.•

Have more questions? Need more answers? Take a look at the Macintosh Q & A Technical Notes on this issue's CD.•

THE VETERAN NEOPHYTE

The Downside

DAVE JOHNSON

According to the menu bar clock on my computer screen, it was 1:38 A.M. My eyes were raw and stinging, my neck was stiff, and my mind was jittery and frazzled. I had to get some sleep soon, because the alarm clock, glowing weakly red in the dark in the next room, right there next to my soundly sleeping wife and animals, was set to go off at 5:30. I'd been ready to stop two hours ago, having lost yet another entire evening to the computer, but I found that there was some obscure system structure that wasn't being disposed of when my program quit, even though I never directly created or used that structure in my code. Dang. I hate that.

The program I was writing is a kind of "magic graph paper" that can help me figure out multiperson juggling patterns. It was originally intended to be the topic of this column, but it took a lot longer to write than I thought it would, and it still isn't done; it will have to wait for some future column. So there I was, deadline approaching, without a topic. I was whining about my foiled plans to my friend Ned (a tolerant listener), complaining loudly about the amount of time and trouble it takes to write the simplest piece of code, bitching and moaning about the trials and tribulations of programming, and wondering out loud what I was going to write about.

Then it dawned on me — write about the downside! Write about the parts of programming that frustrate you so much! Get those nasty feelings out on paper! Purge! Catharse! I could even do another little electronic survey, sort of the opposite of the "Why We Do It" column in Issue 17. Hot dawg.

So that's what this column is about: "What do people hate about programming?" I posted this question to various news groups on the Internet, and sent it out via e-mail to a bunch of programmers I know, and got some good responses. But before I tell you what other people hate about programming, first it's *my* turn, and I'm ready to gripe.

Once upon a time, I loved programming. It was a hobby, something I did for the sheer joy of it, something that was *fun*. I welcomed the chance to learn all the arcane and grungy details of the internal workings of the computer. I binged, ignoring the demands of my home life and my body. I dove willingly into the thick, impenetrable books, joyfully grappling with myriad problems that had nothing whatever to do with the program I was writing. The program itself was in many ways incidental: it was that continual solving of difficult problems that was both the fuel and the reward, the stick and the carrot combined.

Well, I still go on binges now and then, small ones, but it's much rarer. Before, I would pursue just about any harebrained idea that crossed my feverish mind (and abandon most of them later, half constructed, often after I'd enthusiastically programmed myself into a corner). Nowadays I abandon most ideas much sooner, usually long before I even hit the keyboard. Now, every time I think of a fun programming project (which still happens often), I immediately quail at the thought of sitting down and beginning. Knowing up front how much time and effort is required to accomplish even the simplest things just makes me want to go to sleep. Call me a burnout, call me a wimp, call it growing older, or call it growing up: you'll be right on every count.

But why? What changed? It used to be so different. It used to be that I would dive in immediately at the first glimmer of an idea, hacking and slashing my way through the Toolbox with gleeful abandon, forgetting to eat, forgetting to sleep, forgetting to check errors, sitting in the same position for hours on end, staring, typing, staring, typing, staring . . .

Wait a second. *That's* something that bugs me about programming. Even though the action in my head is fast, furious, and fascinating, physically I just sit, stare, and type. Maybe someday I'll get enough RAM in my Duo to actually do some coding outdoors, but I'll *still* be sitting, staring, and typing; I'll just have a better view on the rare occasions I remember to look up from my lap. (Come to think of it, it might be even worse,

DAVE JOHNSON has an inordinate (some say irrational) love of playground swings. He's been a lover of swings and swinging as long as he can remember, and still does backflips out of them from maximum height, impressing mightily any kids who might be watching. He's also been known to suddenly stop the car and leap out at the sight of a swing set he hasn't visited before. No swing shall be left unswung.•

since I'll be forced to use electronic references as well: I won't even get the small breaks that I normally get while flipping pages in some weighty tome of wisdom.) So even though programming is fundamentally a creative thing, teeming with meaning and craftsmanship and beautiful logic, there's a tradeoff. To partake of that sweet creation, you have to be willing to mostly sit still, stare at glowing humming boxes that make your face look green, and type cryptic symbols in a very strict, "filling out forms" kind of way. For hours. And hours. And *hours*. I can almost feel my muscles turning to liquid, my blood slowing and thickening, gravity slowly pulling my withered flesh from my bones.

That's something else annoying about programming, and about computers in general: the amount of time they require. They are infinite time sinks, no doubt about it. Maybe time is just more precious to me now than it was before, and I'm less and less willing to spend it sitting at a computer. Writing software, in particular, *always* takes too long. It takes much longer than it reasonably ought to, and it takes much longer than you think it possibly can. Every time. There's a common rule of thumb for estimating how long it will take to complete a software project: come up with a reasonable estimate of the time required; then double it. Amazingly, even that doesn't do it. I've heard another, more tongue-in-cheek formula that says to take your best estimate, double it, and then jump to the next larger time unit. For example, if you think a project will take 8 weeks, first double it to 16 weeks, then change the time unit from weeks to months: 16 months is reasonable. Now *that* might actually be accurate.

(I remember some time ago there was a seminar here at Apple given by someone who had a system that was carefully worked out to produce accurate estimates of software projects. The system was entirely history based; that is, the estimates weren't based on the details of the project or the estimates of the engineers involved or the marketing plans for the product, but instead were based on how long it had taken to complete past, similar projects. It sounded very promising to me, but you know what? The estimates thus produced were so much longer than those arrived at by conventional means that real adoption of that sort of system would require a fundamental change in the way the software business is run. I haven't heard about it since.)

Part of the reason programming takes so long, of course, is that much of the time is spent on tasks that really have nothing to do with the program you're trying to write, but instead are about bookkeeping, working around the limitations of the machine, trying to figure out how to express your very clear ideas in the hobbled, awkward prose of "modern" computer languages, and so on. That's *another* thing I hate about programming: the mountains of mind-numbing and irrelevant detail you have to wade through and deal with to accomplish the simplest tasks. I don't really give a hoot about the details of the operation of the file system, I just want to get some information onto the disk; I don't really want to know how scroll bars work, I just want the user to be able to navigate an area larger than the window. This is, naturally, a good argument for using frameworks (among many other good arguments), but the promise is still beyond the reality, and even with a good framework there are still mountains of irrelevant detail. They're different mountains, and they might be a little smaller, but they're still mountains, and they're still irrelevant.

Another thing: software is never really done, just shipped. That's another aspect of the "infinite time sink" thing. There's always something more that can be done to make a program better, and there are always bugs that can be fixed. I've heard some artists and writers say that they never actually finish a piece, they just stop working on it (and, incidentally, that knowing when to stop is where a lot of the art is). Software is often the same way.

Programming is also addictive, and I hate that, too. (I'm starting to feel like Andy Rooney here.) It positively *consumes* me. There I'll be in the umpteenth hour, my eyes burning, my head aching, my neck stiff, everyone else fast asleep, warm and cozy in their nice, soft, analog beds. But I can't stop, because there's just one more little wrinkle to iron out, one more small problem to solve. And the solution to that problem leads to another problem, and the solution to that one leads to another, and . . .

Know what else I hate? Bugs. Not the plain old easy to find kind of bugs, but the nasty, subtle, elusive, intermittent kind that just don't seem to have a deterministic cause. They're an unavoidable part of programming, but I hate 'em anyway. (Seymour Papert, in his book *Mindstorms*, made the excellent point that programming is one of the few disciplines where you're *expected* to make mistakes, every time, and an integral part of the process is going back over your work, finding the inevitable mistakes, and fixing them. This is in sharp and healthy contrast to most academic subjects, where mistakes are thought of as unwelcome anomalies rather than an inevitable part of the process. An excellent reason, says Papert, to teach programming to children: it introduces them to the fact that mistakes are an integral part of real-world processes.) Knowing that there really *is* a solution to the bug (and probably an easy one) just pisses me off even more. If there's an

answer, why can't I find it? And after spending days and nights sleuthing my way to an eventual answer, do I rejoice when I arrive? No, I'm just angry that I had to waste so much time on something that didn't really move me further forward. Hope starts to fade; ennui begins its inexorable descent.

And finally, well, programming *hurts*. It hurts my body, and it also hurts my brain. It's unnatural to think like that, composing long strings of imperatives, with no subtlety or nuance or fuzziness of meaning allowed, especially for long, uninterrupted periods of time. It's somehow dehumanizing, because to program well you have to assume the characteristics of the machine, you have to think like one, you have to make your thoughts linear and ordered. It's just not normal.

All right, that's probably enough personal griping. I *do* feel a little better having gotten that off my chest, here in public. Now let's see what other people had to say.

In general I was surprised at the paucity of responses, at least compared to the veritable flood of replies I got when I asked what people *liked* about programming. Of course, I was asking programmers, and since they do it for a living they presumably don't hate it too much.

Of the responses I got, the most commonly hated thing by far was bad or broken tools. This wasn't surprising; programmers *love* to complain about their tools. In particular, buggy compilers were soundly thrashed from all sides as the worst time wasters around. Dealing with your own bugs is one thing; that's a normal part of the programming process. Dealing with bugs in your tools, though, and having to work around them, is something else entirely:

> *I LIKE programming. The only things that bother me are things that are not under my control, like compiler bugs. I hate that.*
>
> — Matt DiMeo

> *After a while, the tools got to me. Tools with bugs and bad interfaces made the day-to-day work more like digging a ditch than the artistic expression it maybe could be. I don't like to dig ditches, I like it to be interesting.*
>
> — Bo3b Johnson, semiretired programmer (the "3" is silent)

This is one of the things that came up over and over: the primitive state of the tools available. Programmers in particular, since they know intimately what the machine is capable of, are appalled at the state of the tools available for programming. Memory management was cited often as a needless hassle — many C and C++ programmers actually mentioned dynamic languages in a positive light, mostly because of the automatic memory management and the banishing of the compile-link-debug cycle.

Lots of folks also complained about the job of programming, and most of those complaints fell into the realm of "the nonprogrammers just don't get it."

> *The interactions with the management. You know, these silly men with ties that say, "Well, you should change that program to act THIS way, and not in the way we agreed last week," when you have the job half done.*
>
> — Maurizio Loreti

Maybe that's why geeks seem to congregate in groups: only other geeks really get it.

Interestingly (from my Macintosh perspective), a number of people mentioned interface programming as something they hated:

> *I have a special hate mode for doing GUI programming. It's boring, it's arcane, and it's ill behaved. Give me systems, give me new real terrain to learn and think about, but leave the GUI programming to robots!*
>
> — Jeffrey Greenberg

This surprised me a little, I must admit, partly because I really *like* that part of programming. Almost all of my little toy projects involve lots of clicking and dragging of widgets on the screen, and now that I think of it, programming didn't really begin to interest me until I discovered the Macintosh and user interaction. But hey, everyone's entitled to an opinion.

Another frequently mentioned offender was lousy APIs:

> *Number two: Poorly designed OS and peripheral interfaces, where I have to keep track of a lot of "moving parts" to do something that should be straightforward.*
>
> — Tom Breton

That's another example of something that seems pervasive: complaints about the software layers that surround most programs these days. It's pretty much unavoidable now; you can't write a program without depending heavily on the software environment you're programming for. Your software is controlled from the outside by other software (typically a GUI these days),

and in turn your software isn't directly controlling the machine like in the good old days, but is instead calling *other* pieces of software (like the Toolbox or a framework) to accomplish its tasks. So naturally it's annoying when the software you depend on is badly designed or buggy. Unfortunately, it's all too common an occurrence.

A few also mentioned a lack of programming quality or a lack of professionalism as a big downside:

> *I get really livid when I find a reckless patch or hack in products, specifically those that make a nightmare for future development and integration. They demonstrate a selfish and irresponsible form of engineering.*
>
> — Dave Evans

> *Unfortunately, most of the programmers I've been around are immature and not well managed, so you end up with these massive schedule and quality problems. I claim it's immaturity, because if we are still stuck in low gear trying to impress our friends with our tricky code, that's high school behavior. That's mostly what I saw. "Ooh, I can save 12 bytes if I write this in a stupid way. Aren't I clever?" Too bad no one can read it. Or, "Ooh, I can make this faster if I write it obscurely. I'm so cool." Never mind that it never gets used. That sort of lack of professionalism is what put me over the edge.*
>
> — Bo3b Johnson

Finally, and near and dear to my heart, is the issue of being forced to interact with the machine:

> *I hate the actual typing in of all the stuff. After a pleasant period of roaming around, scribbling down nice little try-outs and possible solutions, just using old pieces of paper, lying down on the couch, thinking a bit, reading a bit, talking things over with a couple of colleagues, there comes a long, boring period when I'm almost chained to that silly desk, sitting in front of that silly machine, banging that silly keyboard, trying to express my illuminated thoughts in some sort of silly programming language . . . I want my couch, I want my can of beer, I want my cigarettes and my books; THAT'S how I want to program.*
>
> — Jos Horsmeier

> *. . . spending the greater part of my life at a !#@!$* keyboard, staring at a !$&*!# monitor.*
>
> — Anonymous

Yep, that's the part that I hate the most, too; in order to program I have to spend huge amounts of my time sitting at the computer. You know, digging ditches is starting to sound better and better. It's tactile, it's immediate, it's outdoors, it uses muscles beyond those in your forearms, it doesn't consume or pollute your mind, there's no irrelevant detail to deal with, and when you're done you're done. Though I could be wrong, I don't *think* ditch digging is addictive, and I'm quite sure that there's no possibility of subtle and elusive bugs. Sounds good.

Hey, that gives me an idea! With QuickDraw GX's great hit testing and picture hierarchies, I could write a really cool ditch-digging simulator . . .

RECOMMENDED READING

- *Snow Crash* by Neal Stephenson (Bantam Books, 1992).
- *Chicken Soup, Boots* by Maira Kalman (Viking, 1993).

Thanks to Jeff Barbose, Brian Hamlin, Bo3b Johnson, Lisa Jongewaard, and Ned van Alstyne for their always enlightening review comments.•

Dave welcomes feedback on his musings. He can be reached at JOHNSON.DK on AppleLink, dkj@apple.com on the Internet, or 75300,715 on CompuServe.•

Newton Q & A: Ask the Llama

Q *I'm trying to remove indexes that I've added to the names soup. But the code to do it is kind of ugly. First I have to go through all the indexes to see if my index is in the soup. Then if I find the index, I can remove it. Is there an easier way?*

A Yes, there's an easier way. Remember that a call to RemoveIndex will throw an exception if it's passed an invalid index. You can wrap your code in an exception handler and prevent the invalid index exception from leaving your code:

```
ExceptionBasedRemoveIndex := func(theSlotsym, theSoupName)
begin
   local theSoup;
   foreach store in GetStores() do
   begin
      theSoup := store:GetSoup(theSoupName);
      if theSoup then
         try
            theSoup:removeIndex(theSlotsym);
         onexception |evt.ex.fr.store| do
            if CurrentException().error <> -48013 then
               ReThrow();
   end;
end
```

This function will remove a particular index (specified by theSlotsym) from a particular soup (specified by theSoupName) on all currently mounted stores. If the index exists, it will be removed; otherwise, the exception thrown for trying to remove an invalid index will be caught and ignored. If a different exception occurs, it will be rethrown so that other exception handlers or the system can deal with the exception.

Q *I'm writing a utility that is an auto part. My utility needs preferences, but there's no application to add preferences to. Where should I put my preferences?*

A The guidelines for preferences are simple:

- For any addition that has an icon in the extras drawer, the preferences should be part of that application. Use the info button to access them.
- For something that has no icon in the extras drawer, add your preferences to the system preferences roll.

See the sample "Preefer Madness" on this issue's CD for more information.

Q *When text gets pasted into my paragraph view, that text is highlighted. I want to be able to detect when this happens and then be able to unselect the text. How do I do that?*

A When the text gets added, the viewChangedScript will get called with the slot parameter set to **'text**. You can use the SetHilite message to unhighlight the

---

The llama is the unofficial mascot of the Developer Technical Support group in Apple's Personal Interactive Electronics (PIE) division. Send your Newton-related questions to NewtonMail DRLLAMA or AppleLink DR.LLAMA. The first time we use a question from you, we'll send you a T-shirt. •

text. However, the viewChangedScript will get called before the underlying implementation of the paragraph view has been changed. This means you need to call SetHilite in a deferred action.

Q *I'm writing a specialized application for a corporate customer. One of the requirements is that the application launch when the Newton is turned on (a "turnkey" application). Is there a way to do this with the Newton?*

A You can use the installScript of your application to add a deferred action that opens your application:

```
constant kAppSymbol := '|autoLaunch:PIEDTS|;
installScript := func(partFrame)
begin
   AddDeferredAction(func() GetRoot().(kAppSymbol):Open(), []);
end;
```

This will launch your application whenever the Newton is restarted or a card containing your application is inserted. Note that if the application is closed before the Newton is restarted again, the application will not relaunch. Nor will the user be prevented from accessing other features of the Newton such as Names, Dates, or Extras Drawer; that's a much harder problem.

Q *I've been trying to use the protoRoll and protoRollItem to create a roll browser of my own. Everything works fine until I scroll. For a couple of these items I need to tap the down arrow twice for it to go to the next roll item. I see the scrolling view effect, but it just scrolls to itself.*

The height slot in each of the roll items has the same value as the height in their viewBounds slot. If I move the roll items around when they're added to the protoRoll (dynamically from my own protos), they work fine. How can I fix this?

A The problem is that one of the protoRollItems in the items array is larger than the protoRoll. If you make the roll browser larger than the largest roll item, all will work fine; otherwise, you have to scroll the roll item twice to move to the next roll item.

Also, since you imply that the entire large roll item is visible, I assume that the protoRoll has vClipping turned off. If you'd had clipping on, you would probably have noticed that the individual roll item was too large.

Q *I'm having some problems with margins when I'm faxing. A fax without a cover page has different margins than a printed page. The actual viewBounds is the same, but the margins of the fax are different from the viewBounds.*

Also, a fax with a cover page has even different margins. The viewBounds is different, too (20 pixels shorter in height), but that's OK. The problem is that the actual margins when faxed are different from those specified by the viewBounds slot. Is this a known problem?

A Faxes with a cover page have a header line at the top of the fax which takes up those mysterious 20 pixels. In fact, it might be a bug that faxes without a cover

page omit this header, but perhaps the only bug is not documenting that protoPrintFormat (which provides the cover page) also adds that header.

The way to find the correct page bounds is to set the viewBounds of your base print view to that of the parent. The base print view is usually a clView that is a child of a print layout. You can use the following code in the viewSetupFormScript of your base print view to set your bounds to those of your parent:

```
viewBounds := :Parent():LocalBox();
```

Q *I've got an auto part that installs a template for the formulas roll. On the roll item I've got a protoLabelInputLine for data entry, and a button that I want to use to clear the input line. My buttonClickScript is very simple:*

```
buttonClickScript := func()
begin
   SetValue(myInputLine.entryLine, 'text, "");
end;
```

The first time the button is tapped, the input line gets cleared OK; after that it never seems to work, no matter how I code it. Can you help?

A This is a very subtle problem. The answer will be revealed in stages, so that you too can experience the "Aha!"

Observation 1: When you edit the text in any clParagraphView, no new strings are generated. The existing string is destructively modified (excluding the usual _proto copying, of course).

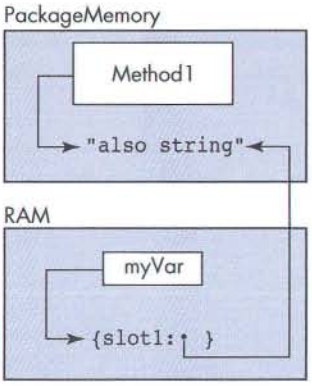
Observation 2: During the compile cycle, the Newton Toolkit will turn all your strings into constants. Contrast this to using braces to construct a frame. As an illustration, assume you have these three methods:

```
Method1 := func()
begin
   return {slot1: "also string"};
end;

Method2 := func()
begin
   return '{slot2: "also also string"};
end;

Method3 := func()
begin
   return "a string";
end;
```

The braces specify a frame constructor. Each time you call Method1 it will return a reference to a newly allocated frame, though not different contents. For example, when the following is executed

```
myVar := call Method1() with ();
```

here's what you get in memory:

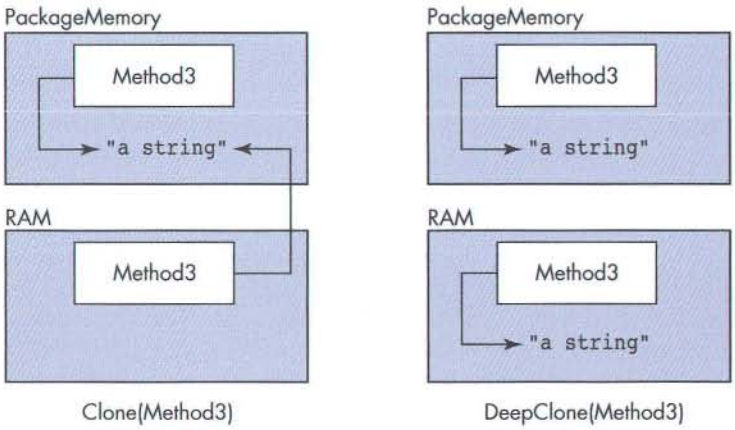


On the other hand, Method2 quotes the frame, which makes it a quoted constant. In other words, each time you call Method2 it will return a reference to the same frame. And Method3 does something else altogether: In the Newton Toolkit, a string is treated like a quoted frame (or array). It's a constant object, so each time you call Method3 it will return a reference to the same string. Note that this means that in both Method1 and Method2 the slot in each frame will reference the same string. Diagrams that show what happens in memory when each of these three methods is executed are provided on this issue's CD along with this Q & A column.

Observation 3: When you call SetValue, you're actually copying the reference to the empty string from your buttonClickScript into the text slot of the entry line. You might think this would cause an error, because the string constant can't be modified. But clParagraphViews are smart: if the string can't be modified (that is, if it's read-only), a copy is made.

Observation 4: I checked in the inspector, and your buttonClickScript is not read-only. This means that the string constant "" in that script is also not read-only.

Observation 5: To prevent the grip of death on a card, you would need to call EnsureInternal on your formula roll entry. This effectively makes a copy of the entire template, including constants, in the NewtonScript heap. The following illustration contrasts a Clone with a DeepClone (which is what EnsureInternal uses). Note that the DeepClone creates a new read/write copy of the string.



Conclusion: You press the Clear Data button once. This sets the reference of the input line string to point to the string constant in your buttonClickScript. Since the string constant is no longer read-only, changing the input line string

destructively modifies the string constant. You may think that this would lead to a bus error or worse, but thanks to NewtonScript, it works as it should. The next time you press the Clear Data button, the input line string reference gets replaced with a reference to the now modified string constant.

The solution is to change the SetValue call to

```
SetValue(dataItem.entryLine, 'text, Clone(""));
```

This will make a copy of the string constant and return a reference to the copy.

Q *Just recently I came into possession of a sword. It was handed to me by a lady in a lake whose arm was clad in the finest shimmering samite. I figure with this sign of divine providence I should be able to wield supreme executive power. What do you think?*

A Fortunately, strange women in ponds have not been used as the basis for a system of government since the Dark Ages. These days supreme executive power derives from a mandate from the masses, not from a farcical aquatic ceremony. If I claimed to be President just because some aquatic gymnast threw a sword at me, I'd be locked up for sure.

Thanks to Don Gummow and our Newton Partners for the questions used in this column, and to jXopher Bell, Bob Ebert, Mike Engber, Kent Sandvik, Jim Schram, and Maurice Sharp for the answers.•

Have more questions? Need more answers? Take a look at PIE Developer Info on AppleLink.•

How're we doing?

If you have questions, suggestions, or even gripes about *develop*, please don't keep them to yourself. Drop us a line and let us know what you think.

Send editorial suggestions or comments to AppleLink DEVELOP or to:

Caroline Rose
Apple Computer, Inc.
One Infinite Loop, M/S 303-4DP
Cupertino, CA 95014
AppleLink: CROSE
Internet: crose@applelink.apple.com
Fax: (408)974-6395

Send technical questions about *develop* to:

Dave Johnson
Apple Computer, Inc.
One Infinite Loop, M/S 303-4DP
Cupertino, CA 95014
AppleLink: JOHNSON.DK
Internet: dkj@apple.com
CompuServe: 75300,715
Fax: (408)974-6395

Please direct all subscription-related queries to *develop*, P.O. Box 531, Mount Morris, IL 61054-7858 or AppleLink DEV.SUBS (on the Internet, dev.subs@applelink.apple.com). You can also call 1-800-877-5548 in the U.S., (815)734-1116 outside the U.S., or (815)734-1127 for fax.

Printing Pains

See if you can solve this programming puzzle, presented in the form of a dialog between Konstantin Othmer and guest puzzler Josh Horwich. The dialog gives clues to help you. Keep guessing until you're done; your score is the number to the left of the clue that gave you the correct answer. Even if you never run into the particular problems being solved here, you'll learn some valuable debugging techniques that will help you solve your own programming conundrums. And please, make KON & BAL's day by submitting a puzzle of your own to AppleLink DEVELOP.

JOSH HORWICH

Josh Hey, KON, where's BAL?

KON Hmmm. That's a good one. Have you checked all the usual places: his cube? the fitness center? prison?

Josh No sign. He won't even return my calls.

KON Maybe his answering machine is on the fritz?

Josh Hold on! Finding BAL was not the puzzle I had in mind.

KON Well, I hope this is an easy one if I have to go it alone.

Josh It's right up your alley. Let's see if all that Sega programming has made you soft. I have a Mac IIci with 8 MB of RAM, a late alpha version of System 7.5, QuickDraw GX beta 3 . . .

KON Hold on, hold on! There's the problem! Swap hard drives with a machine that has working system software, and your bug, whatever it is, goes away. While you're at it, why don't you buy a Mac with a little more horsepower?

Josh Not so easy, KON. We're here to solve these problems, to "learn some valuable debugging techniques," remember? Anyway, I'm printing from Deneba's Canvas to a LaserWriter Pro 630. My machine gets a bus error while spooling a nasty sample document consisting of a bunch of Ferrari F40s that Lance thoughtfully duplicated and rotated in Canvas.

KON OK, let's isolate the offender here. What happens if you install GX beta 3 on the IIci running System 7.1?

JOSH HORWICH (Internet josh@catapent.com) had the rare pleasure of running across this particular bug during the two years he spent on the QuickDraw GX Graphics team at Apple. Now he's working at Catapult Entertainment, Inc., a Cupertino-based company developing what KON affectionately calls a "modem" for home video game consoles. Between Slurpee runs to the 7-11 convenience store and games of pinball, Josh can occasionally be found in front of a logic analyzer, watching a single bit ruin his whole day.•

Josh The problem goes away; the document prints beautifully. You even get all those cool GX printing features, like document redirection and printing extensions. Don't you just love it?

KON It's great! I can't wait to install it. How about some more information about the crash?

Josh What? You haven't figured it out yet? OK, I'll be nice, since BAL is hiding out. Let's install a debugging version of the beta 3 GX Graphics INIT, and see what we can find. I'll be even nicer and give you a version with MacsBug symbols.

KON So where's the crash?

100 Josh It looks like we don't crash in GX itself. MacsBug heap checks reveal nothing amiss in any heap. But we crash in a **CMP.W (A2), D0** instruction, with A2 looking like garbage. What next?

KON How about a **wh pc** MacsBug command to see where we are?

90 Josh The PC is 1270 bytes into a locked, purgeable, relocatable block in the system heap. The block even consists of legitimate code! It's about 16K long, if that's any help to you. A stack crawl reveals no interesting MacsBug symbols, just to make things even tastier.

KON OK, let's try to figure out who owns this block. Find the beginning of the block and use **dm** to look around. Any clues?

Josh Nothing obvious, like the programmer's name and phone number. Only a few cowboys like you would leave such a nice trail. I do notice some four-letter constants near the top, like 'mach', 'fpu ', and 'qd ', but overall the block looks like a bunch of 680x0 opcodes, as one would expect.

KON All right, let's use **il** to look around the block and see if we can find any telltale traps. Maybe from there we can guess what sort of code this is, or even who owns it.

80 Josh Besides the smattering of Gestalts, HLocks, HUnlocks, and GetTrapAddress traps, I notice a _ComponentDispatch and a _SetComponentInstanceStorage call. Overall, this code has very few traps, and lots of computational code.

KON I was told there would be no math! This code sounds like a Component Manager–based code resource that went amuck. Given that we're dealing with printing from GX, I'd guess it's ColorSync and not QuickTime. Let's be skanky and see how we got into this wonderful code. Move the PC to the end of the function, and step us out of here. What do we find?

70 Josh Getting warmer! After walking our way out of here in MacsBug by placing the PC near the end of each function and tracing over the UNLK A6 and RTS instructions, we discover that we are in fact inside a component called by ColorSync! Continuing to step out in this fashion reveals that the trap that was called was _ColorMatch. Didn't you write some of the slime we're looking at now?

KON Nothing doing. It's clearly a GX bug, just like the one from the last Puzzle Page. You GX people like to pawn off your problems on everyone else. What else can you tell me?

60 Josh OK, since I wrote much of the lovely code that has GX calling ColorSync, I'll even lend a hand. Let's restart and do an **atb**

ColorMatch and see what happens. After setting this up, we discover that GX calls ColorSync to convert some colors from RGB to CMYK. The data it passes to CWNewColorWorld looks fine — it's merely the 14-inch Macintosh Color Display color profile. ColorSync returns noErr, and we later crash when we actually try to match a color using CWMatchColors.

KON What version of ColorSync are you running?

50 Josh 1.0.4. It's the one where the code that actually does color matching has been brought native for PowerPC. The folks over in Imaging told me that all they did was massage the code slightly to compile for PowerPC. I hear those IBM compilers are a little stricter than THINK C when it comes to ANSI compliance.

KON Does it work with 1.0.3?

Josh Yep.

KON Hmmm. So what you're saying is we're crashing in ColorSync when printing under GX and System 7.5 to the LaserWriter from Canvas, but it works fine in System 7.1. I'd love to blame the whole thing on 7.5 and call it a day, but the code that dies only makes very standard system calls, which factors the 7.5 code out of the equation. And ColorSync 1.0.3 works. So the problem seems to be with ColorSync 1.0.4. Any other changes for 1.0.4?

40 Josh Since GX relies on ColorSync, we need to know whether it's installed before we install GX and patch out all of the Printing Manager. System 7 loads extensions before INITs in control panels, so I talked the ColorSync guys into making the INIT part of ColorSync live in a separate extension file from the profile picker, which remains in the control panel. Cool, huh?

KON Wonderful. Now the user has twice the chance of throwing the darn thing away, right after getting rid of A/ROSE and DAL. I guess it would be too hard to solve that problem right, and search the Control Panels folder for ColorSync and determine whether or not it's going to load. Now you've created another weird, order-dependent nightmare on the Macintosh. It should give you job security, if nothing else.

Josh Good point, KON. I suppose GX should be clairvoyant and know that ColorSync will load just because it's in the Control Panels folder. Next thing you know, those extension-disabling utilities would be patching the File Manager so that GX's INIT code doesn't find ColorSync when the user disables it.

KON All right, all right. So what does the crashing code look like it's trying to do? Where did this horrible A2 value come from?

35 Josh ColorSync gets this value out of the middle of a relocatable block in MultiFinder temp memory. From the disassembly, my guess is that it's doing a lookup in a hash table of some form.

KON Ah, yes. To speed things up, the matching code remembers recent colors. This way we can avoid a whole lot of math. But why would the block be in MultiFinder temp memory? When ColorSync allocates memory, it first tries the current heap and system heap, and only if there's not enough space in either of those does it allocate the block in MultiFinder temp memory. This seems to imply that you're low on memory.

30 Josh Well, it's just the system heap that's low. Because GX Graphics doesn't want to move application heap memory, it sets the current heap to the system heap before calling ColorSync.

KON It's no surprise that you're low on memory. You have all that System 7.5 garbage floating around in your machine. Tell me more about that block it got the erroneous pointer from.

25 Josh It's 10,054 bytes big, and from the look of things, it's full of trash. I wonder who's ruining it?

KON Let's see. When GX calls CWNewCWorld, ColorSync sets up some memory. Reboot and break on _ColorMatch; once we hit that, break on TempNewHandle. After the TempNewHandle, let's step-spy to see who trashes the location. As long as the block doesn't move, we should find out who's ruining our hash table.

20 Josh A step-spy on a location in a relocatable block? I've got good news and bad news. The good news is that the block doesn't relocate between the allocation and the crash, so the step-spy trick is valid. The bad news is that the step-spy doesn't catch anyone trashing our location.

KON Wait! The location isn't touched *at all*? As in "uninitialized"? How can that be? Right after calling TempNewHandle, I clear out the entire block to 0. What happened here?

15 Josh You're getting warmer! Here's a listing of the code right after TempNewHandle:

```
MOVE.L    D7,-(A7)
CLR.L     -(A7)
MOVE.L    (A3),-(A7)
JSR       *-$3B70
```

KON That looks right. Let's step into the JSR and see what happens.

10 Josh It looks like a simple routine. In fact, it's right out of Symantec's ANSI library:

```
MOVE.L    $0004(A7),D0
MOVEA.L   D0,A0
MOVE.B    $0009(A7),D1
MOVE.L    $000A(A7),D2
BRA.S     *+$0006
MOVE.B    D1,(A0)+
SUBQ.L    #$1,D2
BNE.S     *-$0004
RTS
```

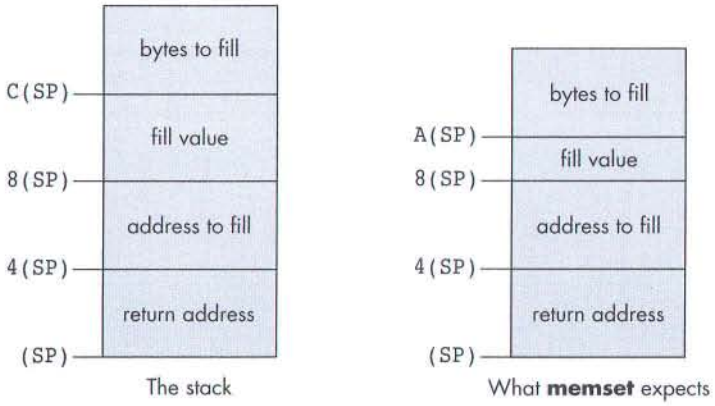
Single-stepping through here reveals that nothing really happens at all. It loads D0 with a pointer to our block, D1 gets 0, and D2 gets 0. It branches to the BNE; then the BNE doesn't loop. Whoops! I bet you wanted to clear a few more bytes than that!

KON How did we end up there? I never even linked with the ANSI libraries back in the 1.0 days! And how did someone screw this up? Let's call up Symantec and scream at them for a while.

5 Josh Not so fast! Let's look at the prototype for **memset**. It can be found in string.h in the C headers folder somewhere deep in the Symantec C++ folder hierarchy. It reads like this:

```
void *memset(void *, int, size_t);
```

It looks like ColorSync thinks that the int is 4 bytes long! After pushing things on the stack, what we've got is what you see on the left here, but **memset** expects the stack to look like what you see on the right. What's wrong with this picture?



KON Of course! The THINK ANSI library comes with the "4-byte ints" option disabled. When taking the matching code native, someone must have decided to make the 680x0 build look as much like the PowerPC build as possible and turned "4-byte ints" on, but didn't rebuild the libraries linked with the code. How does ColorSync 1.0.4 ever work at all on a 680x0 Mac?

Josh Good question, KON! Looking around the TempNewHandle call, we see that ColorSync allocates a handle in one of three ways: with NewHandleClear, with NewHandleSysClear, or with TempNewHandle followed by the call to **memset**. It's being kind by preflighting its memory allocations and choosing a heap only if the allocation would leave at least 32K free afterward. GX is an unknowing partner in crime: it sets the current heap to the system heap before calling ColorSync so that it doesn't inadvertently cause relocatable blocks to be purged or relocated across a GX Graphics call.

KON Rebuilding THINK's ANSI library with 4-byte ints enabled will solve the problem. So how come printing succeeded under System 7.1?

Josh When we printed under 7.5, which had every INIT ever written for the Macintosh installed, and a few MS-DOS TSRs thrown in as well, the system heap was pretty full, so ColorSync tried to allocate the handle in temp memory, using TempNewHandle and **memset**. Crash! Under 7.1, there was lots of system heap space, so ColorSync would just call NewHandleClear and everything would work fine.

KON Nasty.

Josh Yeah.

SCORING

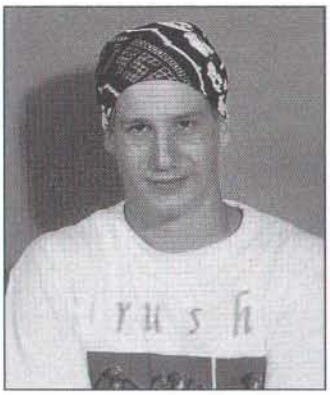
80–100 What a fish story. How big was it?

50–70 Lie this much and you'll end up being BAL's cellmate.

25–40 No fair — this contest not available to the party or parties responsible for the bug in question.

5–20 You're too honest! Don't ever play cards with KON.•

Thanks to Luke Alexander, Tom Dowdy, KON (Konstantin Othmer), and BAL (Bruce Leak) for reviewing this column.•



THE ART OF HUMAN COMPUTING

Finger-Coded Binary

TOBIAS ENGLER

When it comes to the development of Artificial Intelligence systems, scientists tend to conjecture that man himself, encompassing the brain with its bazillion synapses and neurons, is the paradigmatic super-computer. "Computer?" you might ask, "Where are the bits and bytes, the binaries that make it a true computer?" Hold on; disclosure is only a paragraph away.

On a recent camping trip, I observed a friend of mine making some strange gestures with his left hand. When I inquired about this, he asked me whether I could figure out how his raised middle finger represented a binary 4. After getting over my initial surprise at the gesture, I began to understand: it was finger-coded binary (FCB).

THE BASICS OF FCB

The single- and triple-handed excepted, the averagely handed human being is capable of producing a number range of up to 2^10 by using his or her fingers. Every finger represents a bit of information:

Left hand, thumb	(L1): Bit 0, value 1
Left hand, index	(L2): Bit 1, value 2
. . .	
Left hand, little	(L5): Bit 4, value 16
Right hand, little	(R1): Bit 5, value 32
. . .	
Right hand, middle	(R3): Bit 7, value 128
Right hand, index	(R4): Bit 8, value 256 or carry flag
Right hand, thumb	(R5): Bit 9, value 512 or overflow flag

Though I prefer this style (palms facing in), your personal style may vary.

To set a bit, you only have to raise the respective finger; to clear a bit, bend the finger. LSL and LSR do the same they did back in Apple II-land ("shift" your fingers left or right), as do ADD and SUB. (To add, look at each bit in the number you're adding, least significant bit first. If the bit is set, set that same bit in the number you're adding to. If a bit you're trying to set is already set, clear it and set the next higher bit instead.) That's all you need for basic mathematics.

Exercise:	(3 + 6) * 2	
Solution:	Set L1, L2	—> 3
	Clear L2, set L3	—> + 2
	Clear L3, set L4	—> + 4
	"Shift" fingers right 1 position	—> * 2
Result:	L1 clear, L2 set, L3, L4 clear, L5 set	—> 18

Now think up some examples for yourself. Practice hard and daily. Try to outperform your calculator. Then challenge your old Apple II. Then get really brave and challenge your Macintosh. Do it every week. Do it until you excel Excel. When you've gone that far, relax. Congratulations, you're a human supercomputer! Now you can go for the real thing.

CREATING REAL-WORLD APPLICATIONS

In general, nothing is prohibited. (If you're not sure whether showing the police your raised middle finger while shouting "I have to raise four children, you @#$%&!" is legally safe, consult your lawyer.)

In particular, here are some useful examples:

- Tell people your age. They'll envy you for being able to count your age on one hand (you may need two, but that's still amazing).
- Get a kick out of telling your friends your phone number. If you animate your fingers fast enough, you could even compete with QuickTime.
- Earn some extra money. Advertise yourself in any local or national newspaper as either (a) a human binary calculator, (b) a shadow puppeteer, or (c) a lunatic. Finger food obligatory.

I'll keep my bits . . . er . . . fingers crossed for you.

Oh, one last thing: If you want to expand your bit range, you might also use your toes (imagine a gigantic range of 2^20!) or even your ears (if you can wiggle them) or eyes (unless of course you're driving or are otherwise mobile). Good luck!

TOBIAS ENGLER is 19 years old, right-handed, and the "subcaretaker" at a church in Erlangen, Germany, where he's doing his community service (as an alternative to military service). When he's not taking care of anything, he's swimming or playing soccer or badminton, or he's on the road jamming with Rush, Dire Straits, or Bad Religion. •

INDEX

For a cumulative index to all issues of *develop*, see this issue's CD.•

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

RESOURCES

Apple provides a wealth of information, products, and services to assist developers. APDA, Apple's source for developer tools, and Apple Developer University are open to anyone who wants access to development tools and instruction. Developers may access additional information and services through Apple's Developer Programs.

APDA offers worldwide access to development tools, resources, training products, and information for anyone interested in developing applications on Apple platforms. Customers periodically receive the *APDA Tools Catalog* featuring hundreds of Apple and third-party development products. There are no membership fees. APDA offers convenient payment and shipping options, including site licensing.

Apple Developer University (DU) provides training designed to increase your software development productivity. The curriculum includes courses to get you started programming on Apple platforms, as well as advanced, in-depth training on the newest Apple technologies, such as PowerPC, OpenDoc, Apple Guide, and Newton. DU offers courses in Cupertino CA and Portsmouth NH. In addition to classroom training, multimedia self-paced courses and low-cost mini-course tutorials are available.

The Macintosh Associates Program is the primary program for developers using Macintosh technology — including PowerPC, QuickTime, QuickDraw GX, and PowerTalk — who don't want direct technical support from Apple. It's a low-cost, self-support program that also provides a connection with Apple and fellow developers, information on new technologies, and discounts on equipment.

The Macintosh Partners Program is open to developers focused on Macintosh technology. In addition to receiving the same development information and tools as members of the Macintosh Associates Program, Macintosh Partners receive programming-level development support via electronic mail, Macintosh technology seeding, and more.

The Newton Associates Program is a low-cost self-support program for developers who use Newton technology and don't want direct technical support from Apple. It includes discounts on equipment.

The Newton Partners Program is open to developers focused on Newton technology. It offers the same core features as the Newton Associates Program, but also includes programming-level development support via electronic mail, additional hardware purchasing privileges, and marketing programs.

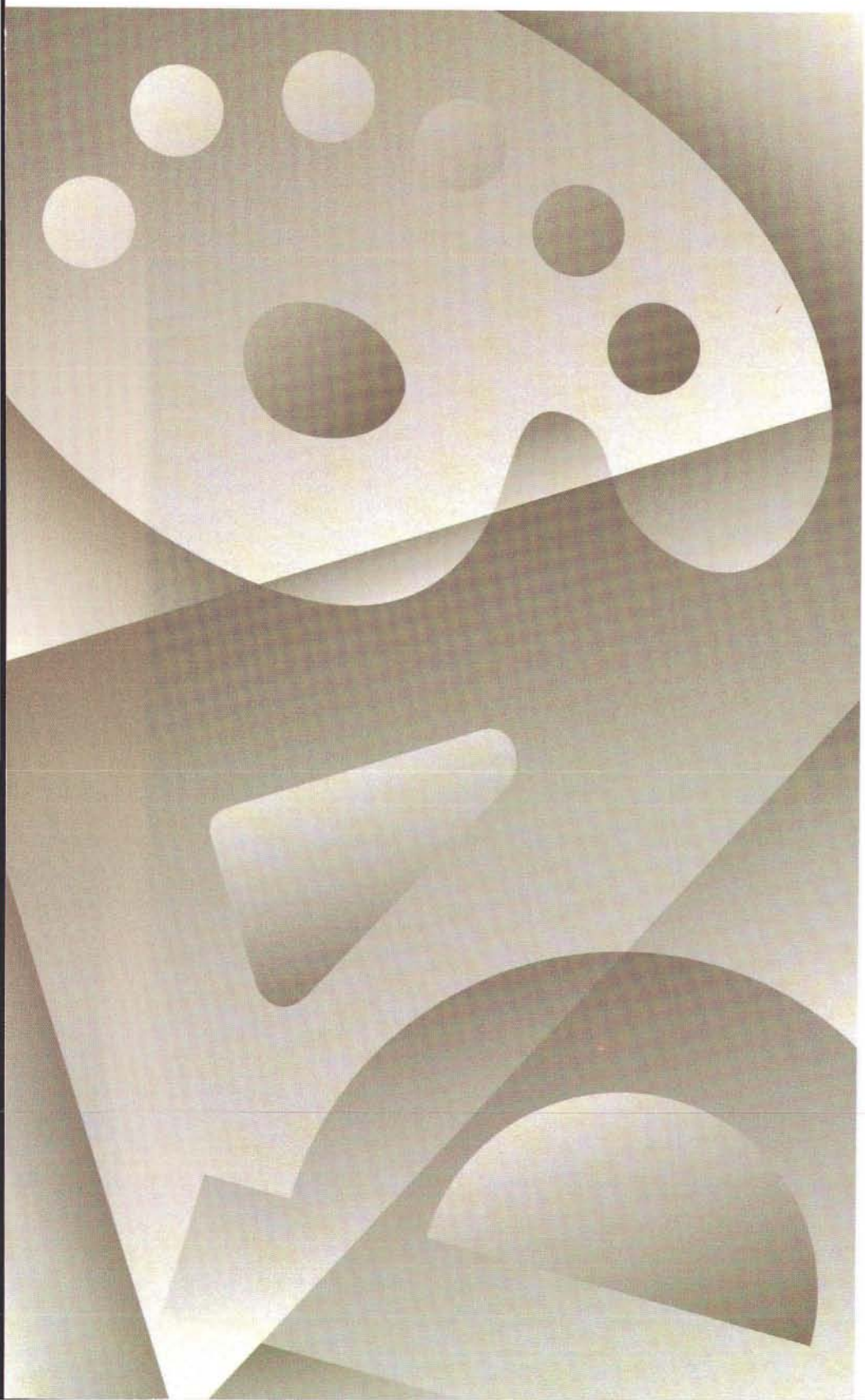
The Apple Multimedia Program is designed for developers interested in the emerging multimedia market. Program features include a quarterly mailing, discounts on third-party products, training, and events.

APDA To order products or receive a complimentary catalog, call 1-800-282-2732 in the U.S., 1-800-637-0029 in Canada, (716)871-6555 internationally, or (716)871-6511 for fax. You can also order electronically (AppleLink APDA; Internet apda@applelink.apple.com; America Online APDAorder; or CompuServe 76666,2405) or write APDA, Apple Computer, Inc., P.O. Box 319, Buffalo, NY 14207-0319.

Apple Developer University Call the registrar at (408)974-4897 to register for a class or receive a current Curriculum Guide and Course Schedule. You can also fax (408)974-0544, send an AppleLink to DEVUNIV, or write Developer University, Apple Computer, Inc., One Infinite Loop, M/S 305-1TU, Cupertino, CA 95014. Self-paced products should be ordered directly through APDA.

Apple Developer Programs Call the Developer Support Center at (408)974-4897, AppleLink DEVSUPPORT, or write One Infinite Loop, M/S 303-2T, Cupertino, CA 95014, for information or an application form. Developers outside the U.S. and Canada should instead contact the Apple office in their country for information about developer programs.

21



Apple Computer, Inc.
One Infinite Loop
Cupertino, CA 95014